Empirical Evaluations on the Cost-Effectiveness of State-Based Testing: Industrial Case Studies and Extensible Tool

Thesis submitted for the degree of Ph.D.

by

Nina Elisabeth Holt

Department of Informatics

Faculty of Mathematics and Natural Sciences

University of Oslo, 2012

Kolofonside (dvs. Copyright- siden, settes av forlaget).

Abstract

Software testing is often conducted as a manual, ad hoc task, as compared to following an automated and more systematic procedure. Consequently, testing is likely to be incomplete and costly to ensure the required level of dependability. Safety-critical software systems must be tested so as to ensure its safe behavior. Despite the importance of being systematic while testing, all testing activities take place, even for safety-critical software, under resource constraints. In order for industry to make the right choices when deciding on how to test their software, more knowledge about how various testing strategies compare in terms of cost-effectiveness is necessary. As thorough software testing is an expensive task, reducing the cost of testing while ensuring sufficient fault-detection effectiveness should be of common interest to industry. Enabling automated testing to check the compliance of implementations against their specifications, model-based testing has become a popular area of research and practice. Test models, for example expressed as UML state machines, describe the expected behavior of the software and provide the basis for systematic and automated generation of test suites. One specific area of research is related to how different coverage criteria of the test models affect the cost-effectiveness of the resulting test suites.

This thesis assesses six state-based coverage criteria and evaluates their cost and fault-detection effectiveness based on 26 real faults collected in a field study at ABB. Eleven of the faults were sneak paths – thus, only 15 of the faults could be killed by the six conformance coverage criteria. Two different test oracles have been applied to compare their cost-effectiveness. Moreover, this thesis also investigates the effect of increasing the test-model abstraction level on the cost-effectiveness of the testing strategies. The coverage criteria were complemented with sneak-path testing. To enable evaluation of the state-based testing techniques, a model-based testing approach, TRUST, was developed and used to automatically generate the studied test suites. Four industrial case studies evaluate each of the testing aspects: coverage criteria, oracles, test models, and sneak paths. The case studies are based on a research project at ABB where a safety-monitoring component in a control system was developed using state machines and implemented according to the extended state-design pattern.

The findings of this thesis include: (1) Development and demonstration of a model-based testing approach based on model transformations. (2) An empirical investigation of the cost-effectiveness of six systematic coverage criteria applied in an industrial project and evaluated by using real faults: all transitions (AT), all round-trip paths (RTP), all transition pairs

(ATP), paths of length 2 (LN2), paths of length 3 (LN3), and paths of length 4 (LN4). (3) A comparison of two oracles: Oracle O1 checks the state invariant of the resulting state in addition to that the current state pointer of the system corresponds to the expected state after the test. Oracle O2 only checks that the state pointer to the current state of the system corresponds to the expected state after the test. (4) An evaluation of the cost-effectiveness when varying the level of details in the test model. (5) A demonstration of the importance of sneak-path testing.

The results show that test suites generated from a precise model according to coverage criteria AT, RTP, ATP, and LN4 when utilizing oracle O1, yields high-quality test suites powerful enough to detect the seeded faults (except from sneak paths). The average cost measured as preparation and execution time were as follows: AT: 5,603 seconds; RTP: 2,731 seconds; ATP: 32,160 seconds; and LN4: 6,145 seconds. LN3 killed 93 percent (14/15) of the mutants at the cost of 645 seconds. Across all six coverage criteria, 88 percent of the seeded faults were detected at 7,905 seconds in preparation and execution time. Applying the weaker oracle O2 at an average decrease in cost around 13 percent, 67 percent of the mutants were killed. By removing details from the test model, the cost of testing was significantly decreased with 85 percent (for both oracles O1 and O2), while only reducing the fault-detection ability by 24 percent for oracle O1 and 37 percent for oracle O2. Note that these results were obtained in spite of a high number of infeasible test cases in test suites generated from the less detailed model as a consequence of wrong test data.

Moreover, the sneak-path test suite detected the eleven remaining mutants that could not be killed by any of the conformance test suites. Thus, the results indicate quite strongly that sneak-path testing is a necessary step in state-based testing as the presence of sneak paths is undetectable by conformance testing.

Finally, this thesis has demonstrated how model transformations can enable state-based testing. The tool was demonstrated to be extensible to support different types of state-based coverage criteria and oracles.

Acknowledgements

First of all, I would like to thank my highly inspiring and motivating supervisors Richard Torkar and Lionel Briand for their brilliant advices, encouragement, help and support. Also a special thank to Erik Arisholm for being my main supervisor from 2009 to 2011. This thesis would not be possible without you!

Furthermore, I would like to thank:

- Dag Sjøberg for encouraging me to apply for PhD.
- Bente Anda for being my main supervisor from 2006 to 2008.
- ABB NOCRC for including me in their project and helping me to recruit subjects to a field study in several of ABB's international departments. In particular, Kai Hansen, Jan Endresen, Knut Asskildt, Sverre Frøystein, and Ingolf Gullesen.
- ABB's research departments in Shanghai, Baden, and Västerås for participating in the field study.
- The Simula Management and Administration, and the Simula School of research for their guidance and support. In particular, Aslak Tveito, Are Magnus Bruaset, Kristin Vinje, and Olav Lysne.
- Hadi Hemmati and Shaukat Ali for excellent cooperation on the development of TRUST.
- My office mates Muhammad Zohaib Iqbal and Amir Reza Yazdanshenas for inspiring discussions.
- Magne Jørgensen, Hans Christian Benestad, Stein Grimstad, Jo Hannay, Gunnar Bergersen, Vigdis By Kampenes, James Dzidek, Audun Fosselie Hansen, Kristin Børte, Aiko Yamashita, Rajwinder Kaur Panesar-Walawege, Tanja Grösche, and Kjetil Moløkken for fascinating discussions and social gatherings.
- The employees at Simula for making a nice work environment.
- Accedo, Genus, and PDMT for being supportive, understanding, and patient.

Great thanks to my very best friends who have supported me throughout these years, and given me priceless memories and environmental change. Special thanks to my dearest Petter Christian Geruldsen and my soul mate Ingrid Schrøen Maudal for their care, support and love.

Last but not least, thanks to my family, especially my mother and father who always back me up, motivate and encourage me. I owe you everything. An extra thank you to my sister

who accompanied me to Shanghai, being very supportive in a difficult period of my life, and to my very best brother for enlivening my life with his brilliant and amusing humor.

Contents

1 Introduction

1.1 Motivation

Today's society highly depends on advanced software, both in private and public sectors. More importantly, as certain uses of software are safety-critical, in the sense that faults in the software may cause serious damage to human beings and physical items, our society depends on software that *behaves as intended*. Examples of safety critical software are the control systems that monitor industrial productions. A typical characteristic of such control systems is that their behaviours depend on the current state of the system.

With software comes the potential of increasing efficiency. Unfortunately, however, along comes the risk of introducing software faults that can cause software to behave undesirable. Both fatal and severe incidents due to software failures during the last decades motivate the need for continuous research on software testing. In short, there is a need for evolving the procedures of software testing.

One approach to software testing derives test cases from a behaviour model of a system, known as Model-Based Testing (MBT) [1]. MBT is not a new domain of research in software engineering [2]. However, in recent years, the level of interest in industry and academia has been rapidly increasing. This interest can be seen from the many academic studies [1, 3, 4, 5, 6, 7] and industrial projects [8, 9, 10, 11] on model-based testing being reported. This suggests that there is an increasing awareness of the benefits offered by MBT [1].

Being one of several possible input models to MBT, state machines are widely used to specify the behaviour of the most critical and complex classes that exhibit state-driven behaviour [12]. State machines are also highly appropriate to facilitate class design; one approach is to follow the state design pattern [13] as demonstrated in [14]. Many object-oriented methodologies recommend modelling components with a state-dependent behaviour with state models for the purpose of testing [12]. This is particularly due to the fact that the specification of a software product can be used as a guide for designing functional tests for the product [15]. As stated by Offutt *et al.* [16], formal specifications represent a significant opportunity for testing because they precisely describe what functions the software is supposed to provide. In particular, such specifications enable automatic generation of test cases using model-based testing tools.

Tool support for MBT has dramatically improved in recent years, but most of the tools specifically target an application context and cannot easily be adapted to others. Many tools have been developed to support MBT [8, 17, 18, 19, 20, 21, 11]. However, all of them have at least one of the following drawbacks:

- They do not support well-established standards for modelling the System Under Test (SUT). This makes it difficult to integrate MBT with the rest of the development process, which in turn makes the adaptation and use of MBT more costly.
- They cannot be easily customized to different needs and contexts. For example, a tester may want to experiment with different testing strategies to help target specific kinds of faults. Furthermore, constraints can evolve, e.g., the test-scripting language in a company can change.

Coverage criteria define how thoroughly the software is tested. The cost-effectiveness of testing significantly depends on the coverage criteria being applied, and constitutes a tradeoff between increasing fault-detection effectiveness and reducing the size of the test suite (affecting costs). Hence, evidence on the cost-effectiveness should be a useful guide for the industry on how to select the appropriate coverage criterion related to cost and criticality of the system under test (SUT). Evaluating the cost-effectiveness of different coverage criteria, however, cannot be performed by analytical means [22]. Empirical studies are crucial to software testing research in order to compare and improve software testing techniques and practices [23].

There are several challenges related to investigating fault-detection effectiveness, amongst others the number of faults to be present in the SUT [22]. Previous work has addressed this problem by seeding artificial faults into correct versions of the SUT using so-called mutation operators [24, 25]. Although results from studies [26, 27, 28] have suggested that faults seeded using mutation operators under certain conditions may be representative of real faults, there is still a need to increase the external validity of results by studying fault-detection effectiveness in more realistic settings, using real faults. A few studies, e.g., [16], partly used naturally occurring faults. However, these constituted only a minor percentage of the total number of faults. Thus, this research was motivated by the lack of empirical evidence in testing software with real faults, and thus complements studies conducted in artificial settings.

Another key challenge in software testing is how to define the test oracle, which automatically provides answers to whether or not the system behaves as intended during test execution. This deserves more research as the cost and fault-detection ability of different oracles may vary substantially [12].

Yet another interesting area of state-based testing, which has been given little attention, is the possible benefits of increasing cost-effectiveness by raising the test model abstraction level. Several studies, e.g. [29, 30], focus on lowering the cost of testing by reducing the size of the test suites, preserving the original coverage. There are conflicting results on how the reduction influences the fault-detection ability of the test suites, in particular with respect to how rigorous the test criteria are. Heimdahl and George [29] thus express a concern for using this technique on structural coverage due to the possible loss of fault-detection. Whereas such test-reduction techniques are based on removing tests in a test suite that do not affect the achieved coverage, this thesis rather focuses on abstracting the test model itself. This means that not only will the cost of testing potentially be reduced due to a lower number of test cases that needs to be generated and maintained, but also due to a less detailed test model that requires less maintenance effort.

1.2 Research Goal

The lack of extensible and configurable model-based testing tools, as well as the need for empirical results to increase the external validity of cost-effectiveness of state-based testing leads to the following research goal: the main goal of this thesis is to empirically evaluate the cost-effectiveness of state-based testing in the context of safety-critical systems by (1) developing an extensible tool for automating the test procedure, and (2) using this tool to empirically evaluate, by means of four industrial case studies, aspects related to the cost-effectiveness of model-based testing.

The first part of the goal includes experiences regarding the design and application of TRUST, a TRansformation-based tool for Uml-baSed Testing. More specifically, the experiences concern:

- The extensibility and configurability in MBT: As motivated in the previous section, most of the existing model-based tools specifically target an application context and cannot easily be adapted to others. There is a lack of tool environments that are extensible and configurable for various application contexts. Therefore, this thesis presents TRUST which is based on model-

transformation technologies and features an architecture with clear separation of concerns and interfaces, thus making it easily extensible and configurable for different context factors such as input models, test models, coverage criteria, test data generation strategies, and test-scripting languages.

- The practical challenges of applying MBT: There are several important concerns that must be addressed when applying MBT, in particular (1) the selection of environment values to increase feasible transitions, and (2) the detection and handling of infeasible test cases due to conflicting guard conditions. Ignoring the former issue may lead to a high number of infeasible transitions due to external variables in guard conditions that never appear with satisfying values; ignoring the latter may also result in a large number of infeasible transitions, though due to a different reason – when combining states from several regions in a state machine, situations may occur where guards on transitions conflict with the source state. This implies that the transition will never be fired.

The studied aspects in the second part of the goal regard:

- State-based coverage criteria: Evaluations of state-based coverage criteria are important in order to provide evidence on how effective the criteria are at detecting faults and at which costs. The following six criteria are addressed in this thesis: (1) all transitions (AT), (2) all round-trip paths (RTP), (3) all transition pairs (ATP), (4) paths of length 2 (LN2), (5) paths of length 3 (LN3), and (6) paths of length 4 (LN4).
- Oracles: Another aspect of model-based testing that may affect the cost-effectiveness is the applied oracle. Hence, two oracles at different abstraction levels are applied in this thesis: (1) oracle O1 checks that the pointer to the current state of the system corresponds to the expected state after the test, in addition to the state invariant of that state, whereas (2) oracle O2 only checks that the pointer to the current state of the system corresponds to the expected state after the test.
- Test model abstraction levels: To identify possible benefits of removing details from the test model on the cost, two different abstraction levels were evaluated in this thesis: (1) a precise model of the system under test, and (2) a less precise model where the contents of every composite state (states that contain other states) were removed.

- Sneak-path testing: The applied coverage criteria provide conformance testing, i.e., checking that the system under test reacts according to the specified behaviour. What is also required, however, is to ensure that the implementation does not include additional behaviour other from what is specified. For this purpose, the system under test is positioned in every possible state and exposed to unexpected, unspecified events. Expected behaviour would be that the state remains unchanged. This approach, which we have applied in this thesis, is referred to as sneak-path testing [31].

1.3 Research Method

In this thesis, model-based testing was evaluated in four industrial case studies conducted in the research department in ABB Norway. The case studies concern a research project in ABB where a safety monitoring component in a safety-critical control system was developed using UML state machines [32] and implemented according to the extended state-design pattern [13]. The extended state-design pattern localizes state-specific behaviour in an individual class for each state, and hence puts all the behaviour for that state in one class. The pattern allows a system to change its behaviour when its internal state changes. This is accomplished by letting the system change the class representing the state of the object. The pattern specifies how and where to implement state and transition actions.

Twelve state-based testing strategies, including the combination of six coverage criteria with two oracles, were compared in terms of cost and effectiveness at two test model abstraction levels. For this purpose, real fault data was collected in a global field study to generate mutated versions of the system under test. The field study included 11 developers from three ABB research departments to solve a maintenance task on the safety-monitoring component. Manual code inspections were used to collect actual faults.

To enable such an evaluation, a model-based testing tool TRUST [33] was developed and used to automatically generate the test suites. To avoid possible randomness in the obtained test results, 30 test suites were generated for each of AT, RTP, and ATP. The reason for randomness in the results is that it is possible to create several test trees that satisfy the same criterion. For the remaining criteria, LN2, LN3, and LN4, the generation of test trees is deterministic. Thus, only one test suite was generated for each of them.

1.4 Contributions

The contributions in this thesis are two-fold: (1) development of an extensible tool that enables automated state-based testing, and (2) four industrial case studies evaluating the cost-effectiveness of model-based testing.

The first part regards the design and application of TRUST, a TRansformation-based tool for Uml-baSed Testing, which can be extended and configured for various application contexts. TRUST is based on model-transformation technologies and features an architecture with clear separation of concerns and interfaces, thus making it easily extensible and configurable for different context factors such as input models, test models, coverage criteria, test data generation strategies, and test-scripting languages.

Using TRUST offers many other potential advantages that are, however, difficult to quantify. For example, it should make the generation of test scripts less error-prone, enable the easy re-generation of test cases when the SUT specifications change, and ensure that testing is systematic and not redundant, an objective hard to achieve for a human tester. In terms of scalability, the only issue seems to be with the flattening of concurrent states, which may take a few hours on complex, highly concurrent state machines. Otherwise, the required processing time involved in TRUST has shown to be, in the worst case, a matter of minutes.

The second part of the contributions was obtained from the four industrial case studies empirically evaluating the cost-effectiveness of state-based testing strategies. The contributions include:

- a field study where real fault data was collected for the purpose of creating mutants to measure the cost-effectiveness,
- a comparison of oracles on different abstraction levels,
- an investigation of the influence of raising the abstraction level of the test model on the cost-effectiveness of the testing strategies, and
- demonstrations of the importance of sneak-path testing for the purpose of detecting unspecified behavior.

Obtained results from the case studies indicate that evaluations of coverage criteria regarding fault-detection are in accordance with results obtained using artificial faults in existing research, thus increasing the external validity of those results.

Applying the more rigorous oracle is proved to be worthwhile as the increase in fault-detection effectiveness exceeds the additional cost as compared to applying the weakest oracle.

Interestingly, removing a rather substantial part of the state machine details resulted in comparable cost-effectiveness as compared to the precise test model. The significant cost reduction (85 percent across all six strategies for both oracles O1 and O2), at an average reduction in fault-detection of 24 percent for oracle O1 and 37 percent for oracle O2, may encourage further research on this type of test-suite reduction.

The application of sneak-path testing stresses the importance of also this type of testing; each of the mutants that remained undetected after applying conformance testing (as expected) were killed by the sneak-path test suite.

In terms of benefits, the comparison of the cost of modeling with the number of test cases generated has shown that using TRUST should yield significant cost savings when applying standard state-machine coverage criteria. In other words, the cost of writing manually the same test cases is likely to be larger than the cost of modeling the system under test (SUT) and generating the test cases.

The findings just presented are relevant both for industry and academia. Research on software testing that is to be adopted in an industrial setting must give evidence of relevance to the industry in terms of being cost-effective. For this, case studies are important in that they give the opportunity to test concepts in realistic contexts. In addition, the effort required by keeping models updated may be easier to accept by practitioners when having evidence of the cost-effectiveness of state-based testing.

Moreover, the research methods used to conduct this study are hoped to prove valuable to researchers who are planning similar studies. Finally, the directions for future work should be useful as guidance in research on state-based testing.

To study the potential for using state-based automated testing, a number of activities were conducted. An overview of these research activities as regards the division of labor can be found in Appendix A.

1.5 Publications

The approach of presenting a PhD thesis as a collection of papers is more and more common. Consisting of numerous parts and pieces, however, this thesis was presented as a monograph as to be able to describe the complete history in a more straightforward manner. Nevertheless,

the four case studies presented in this thesis have been submitted to the journal Information and Software Technology. Chapter 11 has also been submitted to the 23rd IEEE International Symposium on Software Reliability Engineering

Moreover, the development process of the SUT was published at MODELS'06 [14]. Finally, two technical reports regarding the development of the model-based testing tool TRUST [33] and the state-machine flattening component of TRUST [34] were published at Simula as technical reports.

1.6 Thesis Organization

This thesis consists of four main parts: (Part I) introduction, (Part II) development of an extensible model-based testing tool, (Part III) cost-effectiveness evaluation of state-based testing strategies using the prototype described in (Part II), and (Part IV) summary of the thesis. The chapters are organized as follows.

Part I – Introduction

- Chapter 2 introduces the concepts and definitions used in software testing.
- Chapter 3 presents related work as for evaluations of testing techniques.
- Chapter 4 regards limitations in existing research.

Part II – Development of an extensible model-based testing tool

- Chapter 5 addresses the development of the automated testing tool.
- Chapter 6 presents lessons learned and experiences from the development.

Part III – Cost-effectiveness analysis

- Chapter 7 describes the design of the four case studies that present cost-effectiveness analyses of (1) six state-based coverage criteria, (2) two oracles, (3) two test model abstraction levels, and (4) sneak-path testing.
- Chapter 8 to Chapter 11 present results for each of the four case studies.

Part IV – Summary

- Chapter 12 presents lessons learned from applying state-based testing.
- Chapter 13 addresses threats to validity.
- Chapter 14 concludes the thesis.
- Chapter 15 gives directions for future work.

PART I – INTRODUCTION

Part I introduces concepts and definitions (Chapter 2). Moreover, it motivates the study by presenting related work (Chapter 3) and identifies gaps in existing research on cost-effectiveness evaluations of state-based testing (Chapter 4). Chapter 4 also explains how the thesis complements and extends existing research on the cost-effectiveness of state-based testing.

2 Background Concepts and Definitions

This section introduces background concepts and definitions, used throughout the thesis, for software testing concepts and empirical research methods used in software engineering.

2.1 Software Testing

2.1.1 Basic Testing Concepts

According to McGregor and Sykes, *software testing* is the process of uncovering evidence of defects in software systems [35, p. 3]. If a fault exists in the software, it must be exercised to be revealed [31]. The process consists in determining test inputs and expected behavior of the system under test (SUT), executing the test, observing the actual behavior, and finally comparing the expected and actual behavior as evidence of whether the test passed or failed. Debugging and repair of faults, on the other hand, are not considered as testing activities. Important to know is that the SUT does not necessarily have to be a complete system. It can in fact be just a class or a module, but also even a system of systems.

A *test case* consists of an input-output pair, i.e., the input to the SUT and the expected result, which is a "description of the output that the SUT should exhibit for the associated input" [35]. A *test oracle* evaluates the results of a test case, either as a manual, automated or partially automated process. The evaluation itself requires a generator for producing expected results, but also a comparison mechanism to check whether or not the test case passed by comparing actual with expected results [31, p. 918]. A *test suite* consists of a number of test cases, and serves a particular *testing goal*. The goal may be to create a random selection of test cases or to satisfy a specific *coverage criterion* (also known as *adequacy criterion*) that specifies the elements of the SUT to be exercised by a test suite. Different types of coverage criteria are described in Section 2.1.5. If all tests in a test suite together cover 100 percent of

the desired program elements in a coverage criterion, the test suite satisfies that coverage criterion [36] – it is *adequate*.

Software testing is often separated in categories according to the particular source from which test cases are derived. Binder characterizes testing in two main categories: *responsibility-based* (also known as specification-based or black-box testing) and *implementation-based* testing (also known as structural or white box testing) [31, pp. 51-52]. The former type uses specified or expected responsibilities of a unit, subsystem, or system to design tests. The latter relies on source code analysis to develop test cases.

Depending on *when* test cases are generated and executed, testing is known as *offline* or *online*. Offline testing generates tests by using various search algorithms prior to test execution, whereas online testing executes the system to generate the test cases dynamically by traversing the test model according to how the system reacted to the previous input. That is, no test cases exist prior to test execution.

The fault-detection ability of a test suite is referred to as its *effectiveness*, whereas the average testing cost for detecting a fault in a program is the *efficiency* of a coverage criterion [37].

2.1.2 Model-Based Testing

Model-based testing (MBT) is an example of a black-box testing technique. In MBT, test cases are derived from a model of the SUT and/or its environment [36]. Utting *et al.* [36] define MBT as "the automatable derivation of concrete test cases from abstract formal models, and their execution". Models of the SUT are of great value to testing, which enables systematic, focused, and automatic testing [31]. Unlike traditional testing, MBT is known for being a structured, documented, and reproducible approach, which reduces the correlation between the individual test engineers' skills and the test quality [36]. In general, MBT can be divided in five steps, of which Step 1 and Step 2 distinguish MBT from other kinds of testing:

1. Model the SUT and make it test ready [31].
2. Generate abstract tests (sequences of operation calls) from the model. Due to a, in some cases, infinite number of possible tests, coverage criteria are used to say which tests we want to generate from the model. The criteria should be selected according to test objectives and cost.
3. Create concrete tests from the abstract tests, i.e., to make them executable. This includes adding low-level SUT details not present in the abstract test cases. For

example, test data are added as arguments in operation calls, to represent the SUT environment, or as values in guards.

4. Execute the concrete tests on the SUT. However, as the test model contains fewer details than the SUT itself, there is a need to bridge the abstraction gap during execution. An adaptor is a component that maps abstract operations in the test model to concrete API calls in the SUT. The adaptor does not necessarily have to be a separate software component [36] – it is often integrated in the test driver [35], which executes the test cases and collects the results.
5. Analyse test results to provide data on the correctness of the system as aid in debugging. Test cases that fail may be due to a real fault in the SUT or fault in the test case. In addition, false positives can be the reason for a test case to fail. This may for instance be due to unsatisfied guard conditions related to externally controlled variables, implying that test data was not correctly selected.

In this thesis, the MBT approach was applied to state-based testing (SBT) in particular, which will be described in Section 2.1.4.

2.1.3 Modeling Notations in Model-Based Testing

Modeling enables abstraction of system details. This is particularly useful when dealing with development of complex systems, both related to keeping control of components, and communicating with customers and colleagues. In MBT, different kinds of modeling notations are used for the purpose of modeling the behavior of the SUT. Sequences of interactions (like message-sequence charts), mathematical functions (like algebraic specifications), executable processes (like Petri net), probabilistic models (like Markov chains), and data flow modeling (like block diagrams) are all examples of modeling notations as presented by Utting *et al.* [36]. More interestingly, in the context of this thesis, Utting *et al.* [36] differentiate between yet another two modeling notations: state-based and transition-based notations. State-based notations model the SUT as a collection of variables whereas transition-based notations model the SUT as transitions between states. Z [38], B [39], and JML [40] are examples of state-based notations. SmartTesting [41] is an example of a testing tool that supports B. Examples of transition-based notations are finite state machines, Statemate statecharts [42], and Unified Modeling Language (UML) state machines [32]. Rhapsody ATG [19] supports testing from UML state machines.

One commonly used modeling notation in software engineering is UML, which is considered to be the *de facto* formal modeling language [43, 44]. Unlike code, which is a detailed modeling language, and natural language, which easily introduces ambiguity, UML consists of a set of graphical diagrams including use case, activity, class, object, sequence, communication, timing, interaction overview, component, package, and deployment diagrams, in addition to state machines. The diagrams are useful in order to present some part of a model [43]. MBT can make use of several of these diagrams – in particular state machines, class diagrams and sequence diagrams. In this thesis, MBT was based on the state machine, a behavioral diagram type.

Many systems, such as embedded real-time systems [45], telecommunication systems [2, 46], and multimedia systems [47], exhibit state-driven behaviour. Such behaviour can be described in terms of UML state machines that express behavior of the SUT in terms of its observable states and how the SUT changes states as a result of events that affect the SUT [35]. UML state machines, which are extensions of traditional finite state machines, can be used to model such behaviour. The state machine defines allowable transitions and actions as response to the events that may occur. An event may cause different behaviours according to the state of the system at the time the event occurred. Transitions between states may be guarded. A guarded transition implies that the change of state will only occur if the condition specified by the guard is evaluated to true.

2.1.4 State-Based Testing

The different features of UML state machines make it an expressive notation for specifying reactive control systems, and may therefore be used as an aid in state-based testing (SBT). Traditional finite state machines cannot model software systems with concurrent behavior. Concurrency in UML state machines is modeled using composite states with two or more regions [32]. When modeling complex software systems with finite state machines, the number of states and transitions can grow exponentially with system size. This can be handled by UML state machine features for modeling sub machines. Many tools (e.g., [48, 49]) support the modeling of UML state machines.

SBT derives test cases from state machines that model the expected system behavior. The SUT is tested with respect to how it reacts to different events and sequences of events. SBT thus validates whether the transitions that are fired and the states that are reached are compliant with what is expected given the events that are received. States are normally defined by their invariant, a condition that must always be true in that particular state.

SBT has the potential of enabling test automation when executable test cases can be automatically generated from state machines. With automation comes the advantage of reduced effort required by human resources. This regards, not only execution of test suites, but also the generation of test suites and evaluation of results. To automate testing based on UML state machines, test data must be generated to fire triggers associated with transitions, and the triggers typically require parameter values. Test data can be generated randomly from the possible set of values, or using more sophisticated techniques such as constraint solvers [50], or search-based techniques (for example using genetic algorithms for test data generation [51]). Constraints defined on UML state machines, such as state invariants, guards, and pre/post conditions of triggers, should be evaluated during the execution of the generated test cases. As has been shown by many studies, this is a very effective way to detect failures [12, 3]. These constraints are usually written as OCL expressions in the context of UML. Examples of available OCL evaluators are OCLE 2.0 [52], OSLO [53], IBM OCL parser [54], and EyeOCL Software (EOS) evaluator [55].

Test suites derived from test models are commonly known as conformance testing since it checks that the SUT conforms to the explicit behavior model. This is, however, not enough because the state machine may not be completely specified, e.g. event/state pairs may be missing in the state machine. This may introduce sneak paths – bugs that allow illegal transitions (unspecified transitions) or elude guards [31]. In order to address this issue, *sneak-path testing* can be applied. In practice, this is done by placing the SUT in each of the possible states, and then, for each state, trigger every illegal event. The expected behaviour is that the state is unchanged.

In practice, applying every possible value and input sequence in every possible state, commonly known as exhaustive testing [31, p. 52], is not feasible. It is necessary to extract a subset of all possible tests. As we will see next, there are several different coverage criteria that can be used to help select test cases from state machines.

2.1.5 State-Based Coverage

To apply SBT on UML state machines as the input models, several testing strategies are presented in the literature, such as piecewise, all transitions, all transitions *k*-tuples, all round-trip paths, *M*-length signature, and exhaustive testing [31, p. 259]. Being a practically impossible approach, exhaustive testing requires every possible value and sequence of inputs to be applied in every possible state of the SUT, thereby exercising every possible execution

path [31, p. 52]. Therefore, we need to use another testing strategy to make a selection among all possible tests.

Coverage is a measure of how completely a test suite exercises the capabilities of a piece of software [35, p. 85]. There are many types of test coverage. Some criteria are based on covering specific parts of the code structure, such as code coverage (e.g., that certain percentage of the statements must be covered by the test suite), whereas others are related to the specification itself, such as all transition coverage where all transitions must be covered by the test suite.

Previous work on SBT has used coverage criteria that were defined to cover finite-machines. Being an extension to finite state machine, however, the UML state machine can also be tested with those criteria if structures like concurrency and hierarchy are removed [22]. The definitions of the coverage criteria used in the following paragraphs are based on definitions given by Binder [31, pp. 259-266] and Offutt *et al.* [56].

All transition coverage is obtained if every specified transition in a state machine is exercised at least once [31]. The order of the exercised transitions is of no importance. Applying this criterion ensures that all states, events and actions are exercised, and is considered to be the minimum coverage that one should achieve when testing software [31].

All round-trip coverage requires that all paths in a state machine that begins and ends with the same state must be covered. To cover all such paths, a test tree (consisting of nodes and edges corresponding to states and transitions in a state machine) is constructed by breadth- or depth-first traversal of the state machine. The test tree corresponding to the all round-trip strategy is called a transition tree. A node in the transition tree is a terminal node if the node already exists anywhere in the tree that has been constructed so far or is a final state in the state machine. Now, by traversing all paths in the transition tree, we cover all round-trip paths and all simple paths (the paths in the state machine that begins with the initial state and ends with the final state). According to Binder [31, p. 248], this technique will find incorrect or missing transitions, incorrect or missing transition outputs and actions, missing states, and will detect some of the corrupt states.

Another stopping criterion for the transition tree construction is proposed in [22], where a node is terminal if (i) it is a final state of the state machine or (ii) it is a node that already exists on the path that leads to the node. This stopping criterion makes the all round-trip strategy more rigorous, and thus gives more coverage.

All transition-pairs coverage is given by a test suite that contains tests covering all pairs of adjacent transitions. For each pair of adjacent transitions from state S*i* to S*j* and from state S*j* to S*k* in the state machine, the test suite contains a test that traverses the pair of transitions in sequence.

Paths of length n. Sequences of transitions are executed from a particular state. For example, one may include all possible sequences of transitions of length *n* from the initial state.

2.1.6 Mutation Analysis

Mutation analysis is an approach that can be used in the evaluation of the fault-detection effectiveness of testing strategies. It is carried out by seeding automatically generated faults into "correct" versions of the SUT. Generally speaking, only one fault is seeded in each mutant version to avoid interaction effects between faults [22]. Mutants are identified through the static analysis of the source code by a mutation system like [57]. When a test suite detects the seeded fault, we say the test suite has *killed* the mutant. The number of mutants killed by a specific test suite divided by the number of total mutants, referred to as the *mutation score*, is used as a measure of a test suite's fault-detection effectiveness. Some mutants may be functionally equivalent to the correct version of the SUT. These are called *equivalent mutants* and should not be included in the pool of mutants used for analysis. We will follow the same procedure in our study, except for the important fact that mutants will be based on actual faults collected in the field. Having this in mind, un-killed mutants must be analysed and possibly removed from the data that provides the basis for the mutation score.

2.2 Research Methods in Empirical Software Engineering

Software engineering is defined by Sommerville [58] to be "an engineering discipline which is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use". Empirical methods have traditionally been used in social sciences and psychology, fields where it is hard to define formal rules [59, p. 5]. However, as software depends on human effort in order to be produced, making the human behavior a highly important aspect of software engineering, the empirical research methods have proved useful also in the more technical software engineering field.

In empirical research, data is collected by observing or experimenting with the item under study. The data is then used to answer a question or test a hypothesis. Empirical software engineering [59, 60] is research that uses empirical studies to gain knowledge about software engineering. Sjøberg *et al.* presents the following vision for all fields of software engineering: "empirical research methods should enable the development of scientific knowledge about how useful different SE [software engineering] technologies are for different kinds of actors, performing different kinds of activities, on different kinds of systems" [61].

An important aspect of software engineering is that software development is done in a cost-effective and predictable way [61]. To provide such knowledge, the community should seek to develop a scientific approach for software engineering, including research methods, theories, terminology, and a collection of experiences and observations [62]. Results from empirical software engineering should, according to Sjøberg *et al.,* not only be useful to guide the development of new software engineering technology, but also to support decisions taken in the industry. The results are based upon actual evidence, as opposed to theory, and are important input to the decision-making in improvement seeking organizations [59, p. 17].

Experiments, case studies, and surveys are all typical research methods in empirical software engineering [59], and will be briefly introduced in the subsequent sections.

2.2.1 Experiments

An experiment is defined by Shadish *et al.* as follows: "a study in which an intervention is deliberately introduced to observe its effect" [63]. In situations where an investigator can manipulate behaviour directly, precisely and systematically, the experiment is preferred as research method [64, p. 8]. The studied treatments are assigned to subjects at random [59, p. 9], and the objective is to manipulate one or more variables and control all other variables at fixed levels [64, p. 9]. Experiments can be exploratory, descriptive or explanatory, normally conducted in a laboratory setting, which provides a high level of control [64, p. 9]. A common perception, however, is that experiments are often small in size to ensure control of the variables. This may have a negative influence on the external validity of the study. Nevertheless, Dzidek demonstrated the feasibility also of larger sized experiments [65]. The main strength, on the other hand, is according to Wohlin *et al.*, that experiments can investigate in which situations the claims are true, and they can provide a context in which certain standards, methods, and tools are recommended for use [59].

2.2.2 Case Studies

The case study research method is observational by nature [59], and defined by Yin in the following manner: “A case study is an empirical inquiry that investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident” [64]. According to Stake, the purpose of a case study report is “not to represent the world, but to represent the case” [66]. The contribution of case studies is through analytical generalization [63, pp. 341-373, 64, p. 32], rather than statistical generalization, where theories are expanded and generalized, although the motive of a case study may also be a simple presentation of individual cases. Hence, it is important to give a rich description of the contextual factors so that others may relate their organization to the case study results with respect to contextual similarities or differences. Yin states that case studies, like experiments, can be exploratory, descriptive or explanatory. Yin also says that case study as a research method is favoured when there is a “how” or “why” question and when the relevant behaviours cannot be manipulated. Case studies imply lower control than experiments, but high realism [59].

2.2.3 Surveys

Surveys are used to collect answers to questions from a random sample of people in a population [67]. It can be used for many purposes, descriptive, explanatory or explorative [67], often conducted in retrospect, e.g., after some technique has been applied in a company [59]. Typical data collection methods are interviews and questionnaires. Examples of types of information that are collected are attitudes and opinions. Analysis of the responses can be both qualitative and/or quantitative depending on how the data are coded. The use of questionnaires, for instance, allows for collection of highly comparable data due to the format of the answers. An important characteristic of surveys is that the results are generalized to the population [59]. As with any research method, the design must be carefully planned, in particular with respect to the formulation of included questions, and how the sample was drawn from the population.

Having covered important background concepts and definitions, we now move to related work. Chapter 3 presents existing research on evaluating the cost-effectiveness of SBT.

3 Evaluating Testing strategies

This chapter presents related work on evaluating the cost-effectiveness of SBT related to both methodical aspects as well as the obtained results, and motivates the research of this thesis. Although not being a systematic review and hence not a complete overview of related work, Section 3.1 includes the main pieces of related work based on searches using Google/Google Scholar and searches in references. Section 3.2, however, provides a semi-structured literature review from 2009 to 2011 with the intention of picturing state of the art from the most recent years.

3.1 Non-Representative Sample of Related Work

The main reason for not having conducted a systematic review is simply that the start-up of the ABB project that I was involved with already had taken place before I started my Ph.D. studies. There was no time for carrying out a, potentially, time-consuming literature review. Following the project was more important. What is to be presented in this section must therefore be called a non-representative sample of related work.

3.1.1 Coverage Criteria

As one of the first studies on deriving tests from finite-state machines (FSM), Chow presented a testing strategy called *automata theoretic* [2], later known as the *W*-method. It was presented as a "powerful testing tool for checking the correctness of the control structure at the design level of many software systems" [2]. Chow based his method on test sequences derived from a spanning tree generated from the FSM. The method was modified by Binder [31, p. 243] to be used in a UML context – referred to as *round-trip path testing* (RTP). The *W*-method and Binder's adaption first traversed the transition tree to cover all paths, followed by identifying the state that was reached. Binder reused the transition tree from Chow's method, but assumed that it was possible to directly check the state invariant rather than having an identification sequence like Chow. Binder defined the set of paths covered by the tree as round-trip paths as they capture all transition sequences that begin and end with the same state (with no repetitions of states other than the sequence start and end state) and simple paths from the initial to the final state of the statechart. A prerequisite to use Binder's approach is to use flattened statecharts [31, 34] – i.e., all hierarchy and concurrency [68] must be removed.

Among several other similar testing techniques for FSMs that have been suggested, e.g. see the survey and discussions by Lee and Yannakakis [69], the main difference from Chow's approach is the state identification process. The *W*-method and its extensions are the mostly used and studied state-based technique for state-based software testing [12].

Offutt and Abdurazik [15] addressed system level testing by generating test cases from UML state machines. They defined AT, ATP, and full predicate (FP) coverage in addition to the complete sequence for the UML statechart. A test suite that achieves FP coverage ensures that each clause in each predicate on guarded transitions is tested independently [15]. Complete sequence coverage is dependent on a test engineer to define meaningful sequences of transitions to be tested, and hence, this criterion is not automatable or measurable. Moreover, to demonstrate the technique and to evaluate the fault-detection effectiveness of the FP and ATP, they presented an empirical study. The Cruise Control system, developed for research purposes, was seeded with 25 faults and tested with 54 FP tests (reduced to 34 test cases when removing duplicates), 34 ATP tests generated from UMLTest tool (a proof-of-concept test data generation tool), and compared with 27 handmade statement coverage tests. Four of the seeded faults were actual faults, detected during the initial implementation. Results showed that the FP criterion killed all 25 mutants, ATP killed 72 percent (18/25), and the statement coverage tests killed 64 percent (16/25).

Abdurazik *et al.* [70] compared three specification-based testing criteria in an empirical study. The FP, ATP, and specification mutation (SM) test criteria [71] were compared on the basis of a "cross scoring" [72], where tests generated for each criterion are measured against the other. Second, the three techniques were compared on the basis of the number of test cases generated to satisfy them, in a rough attempt to compare their relative costs. A model checker was used to generate tests and to evaluate test sets that fulfilled the selected criteria. The academic Cruise Control system was used in the comparison and mutated with artificial mutation operators. The SM score of the FP tests and the FP scores of SM tests were quite high; in fact the two techniques are relatively similar. However, neither the FP tests nor the SM tests had high ATP scores, and the ATP tests did not have high FP or SM scores. The study showed that ATP tests offer something different from FP and SM tests.

Hong *et al.* [73] presented a test sequence selection method for Statemate statecharts [42] where it was demonstrated that data flow analysis can be applied to the selection of test sequences from statecharts. The method included the transformation of statecharts to extended

finite state machines (EFSMs) in combination with the methods presented in [74, 75, 76] that transforms the EFSM into a flow graph.

A method for deriving test sequences based on the AT criteria while retaining hierarchy in a Statemate statechart was proposed by Bogdanov and Holcombe [77]. The method, which is a modification of [73], was applied in a case study to an aircraft control system provided by DaimlerChrysler Research Laboratory. The method appeared to be applicable to the realistic system, although differences in transition labels between the implementation and the specification caused problems in testing every transition. The implementation had to be used in those situations.

The feasibility and effectiveness of AT coverage was again applied in a case study conducted by Chevalley and Thévenod-Fosse [78]. The criterion had a weakness in that a fault could possibly not be triggered by the particular test case input value for that particular transition. For that reason, statistical test case input values were used. Test cases were automatically generated from UML state machines using a probabilistic algorithm presented in [79]. The principle of the technique was to compensate the criteria weakness by exercising each transition several times. The results were based on 1,559 mutants of an avionics system (6,500 LOC); a mutation score of 91.3 percent was reached (1,423/1,559 mutants were killed). The Flight Guidance System was a research version provided by the Advanced Technology Center of Rockwell-Collins.

The effectiveness of the RTP strategy was later investigated by Antoniol *et al.* [80] in a case study. The RTP strategy was applied in a C++ example program consisting of 450 LOC: two classes and 45 methods. The main class under test was a container class, typical of its kind. Artificial mutation operators were used to seed 44 faults covering 8 mutation operators. The study concluded that the RTP strategy is reasonably effective at detecting faults; 87.5 percent of the faults were detected as compared to 69 percent for random testing. Moreover, their results showed that RTP left certain types of faults undetected, and suggested that by augmenting RTP with category-partition (CP) testing, the fault-detection can be enhanced, although at an increase in cost that must be taken into account.

In a series of three controlled experiments, Briand *et al.* [12] evaluated two variants of RTP testing, and CP testing, in terms of cost-effectiveness, and proposed a way to combine them that was referred to as logical, intuitive, and that could be tailored to the test budget available. Students were used as subjects. The following programs were used in the experiments: (1) a container class from an academic software system, (2) a container class

from a real DNS system, and (3) two control classes from a real DNS system. Two different oracle strategies were compared. Artificial mutation operators were applied in the cost-effectiveness evaluation. Results showed that RTP testing is not likely to be sufficient in most situations as significant numbers of faults remained undetected (from 10 percent to 34 percent), on average across subject classes. This is especially true when a weaker form of round-trip was used where only one of the disjuncts in guard conditions was exercised. By combining RTP with CP testing, however, a large percentage of latent faults could potentially be detected by CP after SBT was applied, yet at significant increase in cost, implying that selection of subsets may be necessary.

To see if the specification-based testing criteria could be practically applied, Offutt *et al.* [16] evaluated the efficiency of state-based test criteria in terms of fault-detection effectiveness and obtained branch coverage. The AT, FP, and ATP coverage criteria were applied in a case study and compared with respect to fault-detection effectiveness and branch coverage. A modified version of the Cruise Control system was mutated using 24 faults (of which 4 were naturally occurring faults). Obtained results showed that 1) the weakest coverage, AT (12 test cases), performed similar to random testing (54 test cases) both in detecting faults and in providing branch coverage (62 percent), 2) ATP coverage (34 test cases) detected 18/24 faults and achieved 75 percent branch coverage, whereas 3) FP coverage (54 test cases) detected 20/24 faults and achieved 83.3 percent branch coverage.

Briand *et al.* [22] presented a simulation and analysis procedure to analyze the cost-effectiveness of statechart based testing techniques, and used this approach to empirically investigate the cost and effectiveness for the most referenced coverage criteria based on UML statecharts: AT, ATP, and FP [15], and a modified version of the RTP coverage referred to as transition tree (TT) [2, 31]. TT is another stopping criterion they proposed for the transition tree construction, where a node was considered terminal if (i) it was a final state of the state machine or (ii) it was a node that already existed on the path that lead to the node. The new stopping criterion made the RTP strategy more rigorous, and thus gave more coverage. Three case studies were used in the evaluation: (1) a container class from an academic example program, (2) the Cruise Control system, and (3) an implementation of a video recorder. The two former studies were real-time systems. Artificial mutation operators were used to create 101, 91, and 139 mutants respectively for the three cases. The following conclusions were drawn from the study: (1) AT did not provide an adequate level of fault detection, (2) ATP detected nearly all faults, but not without an enormous increase in cost compared to AT, (3)

TT was evaluated to be more cost-effective than AT and ATP, although the result depended on two factors: the extent to which guard conditions were present in the statechart, and the extent to which the transition tree captured realistic and meaningful usage scenarios, and 4) FP was as effective at revealing faults as ATP, yet more expensive.

A study by Briand *et al.* [81] was conducted that aimed at investigating how data flow information could be used to improve the cost-effectiveness of state-based coverage criteria when more than one tree existed. Two case studies were carried out: (1) the Cruise Control system was inserted with 91 faults using artificial mutation operators, and (2) 131 mutants were generated from an implementation of a video recorder. Results showed that data flow information was useful for selecting the most cost-effective transition tree. They found that the transition tree that contained the largest number of du pairs or definitions would be the most effective at detecting mutants. A more optimal RTP strategy was thus proposed, including (1) identifying the tree that covers the largest number of definitions, and (2) complementing it by tree paths from the other trees that cover definitions not already covered by the initial tree. Note, however, that there were neither guards on transitions in the Cruise Control case, nor parameters in events. When applying the RTP on state machines without guards, the coverage will in fact only cover all transitions.

AT and ATP coverage criteria were compared to mutation-based criteria by Paradkar [82]. The study reported that mutation-based testing had higher fault-detection effectiveness, but at a higher cost than the structured criteria.

Paradkar [83] also conducted an empirical study on fault-detection effectiveness and cost (in terms of size) where FP coverage, BZ-TT [84], mutation based testing, and user defined test objectives were compared. The BZ-TT method concerns the operations in a model and generates tests with boundary inputs when the system is in a state where at least one state variable has a minimum or maximum value [83]. Two case studies were conducted: (1) the ATM application – a simple academic system (eight classes, 153 mutants), and (2) the VM application (one class, 56 mutants). It was found that the mutation based technique provided the best fault-detection effectiveness, followed by BZ-TT and FP. FP was more useful for an application which had a complex guard condition.

Mouchawrab *et al.* [85] addressed the impact of using statecharts for testing class clusters that exhibit a state-dependent behavior, and reported on a controlled experiment that investigates the effectiveness of SBT using RTP when compared and combined to white-box, structural testing. The experiment involved 48 students who were assigned to generate tests

for the OrdSet example class, and the Cruise Control system using RTP and block and edges coverage. Results could not find differences in the fault-detection effectiveness of the two strategies. Combining the strategies, however, proved to be significantly more effective. The fault detection effectiveness was found to vary to a large extent depending on how precisely the statechart described the behavior of the software under test. A request for more research was proposed in order to provide clear and precise guidelines regarding when to use statechart-based testing and how to integrate it with other testing strategies.

An overview of the most relevant work is summarized in Table 1. In terms of empirical evaluations, the most studied state-based coverage criteria were FP, ATP, RTP, and AT. The FP criterion tends to kill higher or similar number of mutants as ATP, although at higher cost. With this in mind, the ATP, RTP, and AT coverage criteria were selected for being studied in this thesis. When also considering the experimental setting, we see that only two of the nine studies were executed in industrial settings. Of these two studies, the first study [77] did not report the nature of the seeded faults, whereas the second study [78] reported the use of mutation operators (artificial faults). The two studies [15, 16] that actually did report use of real faults (yet only 4/20 and 4/21 of the faults were real) were conducted in laboratory settings. A clear conclusion can be drawn regarding the nature of the seeded faults; there is a lack of empirical studies applying real faults in the evaluation of testing strategies executed in an industrial context.

Table 1 Overview of related work on coverage criteria evaluations

Reference	Objective	Research Method	Context	Applied Testing strategy(ies)	Seeded Faults
Offutt and Abdurazik [15]	To take advantage of UML to produce highly effective software system-level tests. They define four state-based coverage criteria for generating test data from formal state-based, develop a tool, UMLTest, and demonstrate system level testing by generating test cases from UML statecharts using UMLTest.	Empirical evaluation	Laboratory	AT, FP, ATP, and the complete sequence for the UML statechart. Only FP and ATP were evaluated.	21 artificial, 4 real faults
Abdurazik *et al.* [70]	To compare three specification-based testing criteria in an empirical study.	Empirical evaluation	Laboratory	FP, ATP, and specification mutation test criteria	Artificial
Bogdanov and Holcombe [77]	To propose a method for deriving test sequences based on the AT criteria, while retaining hierarchy.	Case study	Industry	AT	N/A
Chevalley and Thévenod-Fosse [78]	To evaluate the feasibility and effectiveness of AT coverage using statistical test case input values.	Case study	Industry	AT	Artificial
Antoniol *et al.* [80]	To assess the effectiveness of RTP, and to determine whether it could be complemented by other testing strategies at some additional cost.	Case study	Laboratory	RTP	Artificial
Offutt *et al.* [16]	To see if the specification-based testing criteria could be practically applied, to make a preliminary evaluation of state-based test criteria with respect to fault-detection	Case study	Laboratory	AT, FP, ATP, and the complete sequence for the UML statechart.	20 artificial, 4 real faults

	ability and branch coverage.			Only AT, FP, ATP were evaluated.	
Briand *et al.* [12]	To evaluate two variants of round-trip path (RTP) testing, and CP testing, in terms of cost-effectiveness, and propose a way to combine them. Two different oracle strategies are compared.	Three experiments	Laboratory	Two variants of RTP, and CP testing	Artificial
Briand *et al.* [22]	To present a precise simulation and analysis procedure and to empirically investigate the cost-effectiveness of test case selection strategies based on UML statecharts.	Three case studies	Laboratory	AT, RTP (TT), ATP, and FP	Artificial
Paradkar [82]	Compares AT and ATP to mutation based criteria	Case study	Laboratory	AT, ATP, and mutation based criteria	Artificial
Briand *et al.* [81]	Investigate how data flow information can be used to improve cost-effectiveness of state-based coverage criteria.	Two case studies	Laboratory	RTP	Artificial
Paradkar [83]	Evaluate fault-detection effectiveness and size of selected model-based testing techniques.	Two case studies	Laboratory	FP, BZTT, mutation based testing, and user defined test objectives	Artificial
Mouchawrab *et al.* [85]	Investigate the fault-detection effectiveness of UML statechart-based testing when compared and combined to white-box, structural testing.	Controlled experiment	Laboratory	RTP and white-box testing	Artificial

3.1.2 Oracle Comparisons

Both in practice and research, much focus in today's software testing is related to the generation of test inputs. Furthermore, the important aspect of test oracles, which evaluates whether or not the software under test executed as expected, has received little attention. Staats *et al.* [86] claim that by investigating oracle selection in combination with test inputs, especially how the two aspects complements or influence each other, improvements in the efficiency of testing may be achieved. Studying different types of oracles is an important area as it is not a feasible solution to monitor everything due to the high cost of creating and maintaining such an oracle [87].

Furthermore, Staats *et al.* [88] addressed the lack of a common framework for empirical testing research. They provided such a framework, focusing on problematic areas in today's research as the assumptions about research results on fault-detection ability due to missing focus on the relationship between structure of system under test, testing strategy and oracle. The absence of such a focus may lead to results that can be misleading, cannot be generalized, or that do not facilitate comparisons of strategies.

Applying structured SBT criteria implies a large number of tests, which makes manual evaluation of results an impossible task. It is thus a prerequisite that the test oracle is automated as to consider SBT of real software a feasible approach.

Manual testing, on the other hand, usually results in smaller test suites that enable manual inspection of results. There are several types of oracles to be used in SBT, e.g. checking the abstract state (state invariant) [31], checking the concrete state (i.e., checks all attribute values) [31], and checking pre and post conditions for operations and class invariants [89]. To the author's knowledge, however, only the study of Briand *et al.* [12] has compared various oracles in SBT strategies. Two different oracles were compared in terms of fault detection and cost. The first oracle was a precise oracle that checks the concrete state of objects; the other checks the state invariant (abstract state). They found significant differences between the two oracle strategies, which emphasizes the importance of choosing the appropriate oracle. In other fields, there are several studies that address this aspect, e.g., GUI testing [90, 91] where results reveal that employing expensive oracles leads to the detection of more faults using relatively few test cases.

3.1.3 Test-Suite Reduction

Reducing the test-suite size by abstracting the test model is yet another area of related work where few studies have been carried out in the context of SBT. Heimdahl and George [29] found that the size of the specification-based test suites can be dramatically reduced and that the fault detection of the reduced test suites is adversely affected. Wong *et al.* [30] investigated the effect on fault-detection of keeping block and all-uses coverage constant while reducing the size of a test suite. They found that effectiveness reduction was not significant even for the most difficult faults, which suggests that minimization of test suites can reduce the cost of testing at slightly reduced fault-detection effectiveness.

3.1.4 Sneak-Path Testing

Well-known state-based testing strategies like all transitions, all transition pairs, and round-trip paths [31] seek to compare explicitly modelled behaviour to actual software execution. However, it is also important to test whether or not the software handles unspecified behavior, so called sneak paths [31], in a correct way. State machines are usually incompletely specified and this is normally interpreted as events on which the system should not react, that is changing states or performing actions. Sneak-path testing sends every unspecified event in all states. In other words, sneak-path testing aims to verify the absence of unintentional sneak paths in the software under test as they may have catastrophic consequences in safety critical systems.

Investigating the impact of round-trip path (RTP) testing on cost and fault detection when compared to structural testing, Mouchawrab *et al.* [92] conducted a series of controlled experiments. The study was a replication of [12] where one of the findings was that not testing transitions to self resulted in many faults not being detected. Hence, in the replication experiments they extended the testing strategy by complementing the RTP criterion with sneak paths as recommended by Binder. Results showed that sneak-path testing clearly improved fault detection. The collected data thus strongly suggests complementing RTP with sneak-path testing. No other empirical study evaluates the testing of sneak paths and there are no studies in realistic industrial contexts.

3.1.5 Tool Development and Practical Evaluations

Several well-known, model-based testing tools have been developed in recent years, such as TDE/UML (Siemens) [8], SpecExplorer (Microsoft) [11], and IBM Rational Functional Tester [10]. Based on just three sources, we were able to find references to more than 50 model-based testing tools. In this thesis we focus on configurations of TRUST with state

machines and thus we are interested in comparing this SBT version of TRUST with other SBT tools. Consequently, we focus our discussion of related work to tools that (i) are (partly) based on UML state machines, (ii) automatically generate executable test cases including test oracles, and (iii) have at least some support for extensibility and configurability.

After applying these criteria on more than 50 model-based testing tools [1, 9, 93], we were left with five tools [21, 17, 18, 19, 94]. We then collected information regarding the extensibility and configurability of their different features (input model, testing strategy, and output language of the tools). Since TRUST generates test cases from UML state machines, the following information was collected related to UML state machines from the tools to determine the degree to which their input model can be extended and configured:

- UML metamodel: As the UML metamodel undergoes changes on a regular basis, a test tool must have the ability to accommodate these changes with reasonable effort.
- Constraint evaluation: A UML state machine may contain various types of constraints such as state invariants and guards. These constraints can be defined in different languages such as OCL, Java, or any other tool-specific language. Therefore, the tool architecture should easily accommodate changes in constraint language or evaluation technology.
- Support for UML profiles: UML profiles provide an extension mechanism to support modeling for particular domains and platforms, for instance the MARTE profile for modeling real-time and embedded systems [95]. An extensible tool should be able to accommodate models based on different UML profiles.

The last part of the analysis considered the tools' extensibility and configurability regarding test models and coverage criteria, test data generation techniques, and test-scripting languages since these are the components of a testing strategy. We will now provide a summary of our analysis regarding the extensibility and configurability of these five tools.

Conformiq Tool Suite [21] is an eclipse-based tool used to generate test scripts from system models specified in QML (Conformiq Modeling Language). This language is based on UML and Java/C# compatible syntax and is supported by a tool called Conformiq Modeler [21]. Conformiq Tool Suite can be configured for the following state machine coverage criteria: state coverage, transition coverage, 2-transition coverage, all paths coverage, and implicit consumption (criterion to check that system ignores transitions that are not explicitly defined on a state), branch coverage, atomic condition coverage, and boundary value pattern

(to cover boundaries of decisions in guards) [21]. Conformiq Tool Suite supports extensibility by means of plug-ins, which can be coded in C++ or Java. The plug-ins can be written for changing the test-scripting language, logging formats, and test execution type (online vs. offline test execution). This means that Conformiq Tool Suite can only be configured for predefined coverage criteria and cannot be extended to additional test models, coverage criteria, and test data generation other than what is already provided. However, Conformiq Tool Suite can be extended for different test-scripting languages by implementing specific plug-ins.

The tool Automatic Test Generator/Rhapsody (ATG) [19] is a module of I-Logix STATEMATE and Rhapsody products. ATG can be configured to generate test cases from models based on a set of coverage criteria such as state and transition coverage and modified condition/decision coverage on guards in state machines. ATG doesn't provide any extension mechanisms, however.

AGEDIS is a tool for automated model-driven test generation and execution for distributed systems. It has been made usable and interoperable with external tools by defining clear external interfaces in the tool. In addition, well-defined internal interfaces make AGEDIS more reusable. For instance, the user can define or select a coverage criterion through a test generation directives interface, which includes coverage criteria, constraints (which are additional criteria) on the test suite, and test purposes [96].

There is also a defined interface for abstract test cases that makes the tool open for later extensions to other test-scripting languages than TTCN-3. However, most of these interfaces are not defined by standardized well-known languages. For example, standard OCL constraints on an input UML model should be transformed into their interface language (IF) format [18].

ParTeg (Partition Test Generator) [94] is a test generation tool dealing with the reuse of state machines for automatic test case generation in the context of product lines. Regarding configurability, ParTeg allows the user to choose from a set of coverage criteria: state coverage, decision coverage, and modified condition/decision coverage as well as boundary value coverage criteria for input data to cover boundaries of decisions in guards. No attempt was made for making it extensible and configurable with respect to input models, test data generation, and output languages.

MOTES [17] is a model-based testing tool for generating TTCN-3 tests. MOTES accepts Extended Finite State Machines (EFSM) as input and requires test data to be prepared manually before the test case generation phase. However, it provides some extensibility

opportunities for output models by having a standard input interface. For example, UML state machines created using third party CASE tools (for example Poseidon [97]) can be imported to MOTES, although state machines must be flat without concurrent and hierarchical states. MOTES provides a configurable set of coverage criteria such as selected states, selected transitions, and all transitions.

Table 2 gives a summary of the abovementioned tools regarding their extensibility and configurability for respectively the input model, testing strategy, and test-scripting output language. A clear conclusion from the summary is that current tools usually support configurable coverage criteria (within a limited set) and three of them are extensible regarding new output languages. However, hardly any SBT tool provides support for extending it to new input models or test data generation strategies. Therefore, we were encouraged to develop an extensible and configurable tool with well-defined interfaces and simple extensibility mechanisms. The requirements and development of our tool will be introduced in Chapter 5.

Table 2 Extensibility and configurability of the current SBT tools

Tool name	**Input model**			**Testing strategy**		**Test script output language**
	UML metamodel	**Constraint evaluator**	**UML profiles**	**Test model and coverage criteria**	**Test data generation**	
Conformiq Tool Suite	-	-	-	Configurable	Configurable	Extensible and configurable
ATG	-	-	-	Configurable	-	-
AGEDIS	-	Extensible	-	Extensible and configurable	-	Extensible
MOTES	-	-	-	Configurable	-	Extensible
ParTeg	-	-	-	Configurable	-	-

3.2 Semi-Structured Literature Review from 2009 to 2011

In order to capture recent work in the field of state-based testing, a semi-structured literature review was conducted from the years 2009 to 2011.

3.2.5 Research Method

As the purpose of this review is to illustrate an example of state of the art, papers were extracted from a selection of what can be considered as the three most relevant journals and conferences for software testing (please refer to Table 3).

Table 3 Selection of journals and conferences

Publication type	Database	Publication
Journal	ACM	Transactions on Software Engineering and Methodology (TOSEM)
Journal	IEEE	Transactions on Software Engineering (TSE)
Journal	Wiley Online Library	Software Testing, Verification, and Reliability (STVR)
Conference	ACM	International Symposium on Software Testing and Analysis (ISSTA)
Conference	IEEE	International Conference on Software Testing, Verification and Validation (ICST)
Conference	IEEE	International Symposium on Software Reliability Engineering (ISSRE)

The following search strings were used as inclusion criteria: “state-based testing”, “state based testing”, “state-machine testing”, “state machine testing”, “model-based testing”, and “model based testing”. To be included in the selection of papers, the search strings had to be present in the title or in the abstract. Also, the search string “UML” had to be found in the paper. Furthermore, the criterion for publication year was set to the year range from 2009 to 2011.

Papers that did not include empirical studies were excluded from the sample.

3.2.6 Results

By executing the searches described in the previous section, 24 papers were found. Only six of these, shortly introduced in the following paragraphs, however, appeared to be relevant to this thesis. Only one paper (though not relevant) was returned from the search in TOSEM

executed on the ACM database. Executing the searches in IEEE for TSE publications resulted in one relevant paper. Twelve papers were found among STVR publications in the Wiley Online Library, of which two were included. Continuing with the conferences, no papers were found in ACM when restricting the search to ISSTA publications. Eight papers, of which one was relevant, were returned from searching in ICST publications (IEEE). Finally, two papers were found among the ISSRE publications (IEEE) of which both were included in the review. Please refer to Appendix B for search results per publication, both numbers of excluded and included papers.

Investigating the impact of state-machine testing (the round-trip path coverage criterion in particular) on fault detection and cost when compared with structural testing, Mouchawrab *et al.* [92] conducted a series of controlled experiments. Results showed that there was no significant difference between the two strategies regarding fault-detection effectiveness. Combining the two strategies, however, yielded significantly more effective results.

The round-trip path criterion was further studied by Briand *et al.* [98] in the context of UML state machines with focus on how to improve the criterion's fault-detection effectiveness. They investigated how data flow analysis on OCL guard conditions and operation contracts could be used to "further refine the selection of a cost-effective test suite among alternative, adequate test suites for a given state machine criterion" [98]. A methodology on how to perform data flow analysis of UML state machines was presented. Results from two case studies suggested that data flow information in a transition tree could be used to select the tree with the highest fault-detection ability.

In [99], Khalil and Labiche addressed the assumptions about the round-trip path strategy regarding the equivalency of exercising paths in the tree that do not always trigger complete round trip path versus covering round-trip paths. They investigated the consequences of the assumption not being held in practice. Finally, they proposed yet a new algorithm for generating the transition tree, which resulted in higher efficiency and lower cost.

From the perspective of executing MBT in practice with respect to limited time and resources, three papers on similarity-based test selection address the problem of large test suites that are automatically generated by MBT-tools. Addressing the topic of scalability with respect to large test-suite sizes when applying model-based testing in practice, Hemmati and Briand [100] investigated and compared possible similarity functions to support similarity-based test selection. Empirical data on the most cost-effective similarity measure was collected by applying the proposed similarity measures and a selection strategy to an industrial software system. Results from the case study showed that using Jaccard Index to

measure the similarity of the test cases (which were represented as a set of trigger-guards) of the respective test paths obtained the best results in terms of cost and effectiveness. They reported a significant reduction (77 percent) in test execution cost.

Continuing the work presented in [100], but this time trying to gain insights into why and under which circumstances a particular similarity-based selection technique can be expected to work, Hemmati *et al.* [101] investigated the properties of test suites with respect to similarities among fault revealing test cases. They conducted experiments based on simulation where two industrial case studies were used to guide the simulations. Obtained results confirmed their assumptions that similarity-based test case selection would perform better when "test cases which detect distinct faults are dissimilar and test cases that detect a common fault are similar". They also found that similarity-based test case selection is less effective in cases when a small group of transition paths is mostly disconnected from the rest of the state machine.

Having a motivation similar to Hemmati and Briand [100], Cartaxo *et al.* [102] also addressed the problem of large test suites. A test case selection strategy was compared with random selection by considering transition-based and fault-based coverage. Based on results from three case studies, they found that the similarity-based test case selection can provide more effective test suites than random selection.

3.2.7 Summary

As we have seen, there is a clear gap between existing studies from the three recent years (2009 to 2011) as compared to the work presented in this thesis. To summarize the findings of the semi-structure review, [92] compared the round-trip path coverage with structural testing in terms of cost and effectiveness, [98, 99] reported on attempts on improving the round-trip path coverage, and finally, [100, 101, 102] addressed test-suite size reduction using similarity-based test selection. The latter three studies may not be highly relevant for this study. Yet they were included due to their practical use in potentially reducing the number of test cases generated by the coverage criteria.

None of the included papers, however, compares cost and effectiveness of AT, RTP, ATP, LN2, LN3, and LN4 when varying the oracles and test model abstraction levels by using mutation testing with real faults.

Based on the related work presented in this chapter, Chapter 4 seeks to motivate the research in the thesis by identifying gaps in existing research.

4 Limitations in Existing Research

This chapter seeks to identify areas in existing research presented in Section 3.1 and Section 3.2 that need further exploration, providing a detailed motivation for the research in this thesis.

Although a growing number of studies address the fault-detection effectiveness of state-based test criteria, very few studies have been conducted on realistic industrial software – in particular, studies that evaluate state-based testing strategies using real faults on industrial programs. The academic SUT that most studies have used in their evaluations is the well-known *Cruise Control* system [103, p. 595]. However, the example is considered to be small in size as the flattened specification only contains six states and 19 transitions. Even though the Cruise Control system is known for being a typical control system, there may still be issues in other systems not present in the frequently used academic system. The convenient reuse may thus be a threat to the external validity of the obtained results. Future research should aim at evaluating coverage criteria by varying the SUT instead of using the same SUT over and over again.

Moreover, surveying existing research shows that extremely few studies apply real faults when using mutation analysis for evaluating various testing strategies – the use of artificial faults is prevalent. As stated by Andrews *et al.* [28], a problem when evaluating testing strategies is that real programs with real faults are rarely available. Except from the two studies of Andrews *et al.* [28, 26], in addition to a few studies where only a small percentage of the seeded faults were real [16, 15], artificial faults are used in the reported testing strategy evaluations [70, 77, 80, 12, 22]. As a consequence, little is known about how such structured test approaches compares in detecting real faults. In this thesis, however, the comparison is exclusively based on real faults that were collected during a field experiment at three ABB departments, during fall 2008.

Like in the studies just mentioned, this thesis investigates the *effectiveness*, i.e., the fault-detection ability, of SBT criteria. Moreover, this thesis also addresses the *cost* of SBT. Only a few of the presented studies report the cost of SBT. Several aspects of cost are presented in this thesis:

- the cost of modeling,
- the cost of generating and executing test suites that satisfy six state-based coverage criteria, two oracles at two different test model abstraction levels,
- practical issues regarding test execution, and

- tool development.

Furthermore, this thesis explores the use of two test oracles with different precision levels.

Also, the lack of research on the interesting aspect of increasing the test model abstraction level motivates this thesis. The studies addressed in Sections 3.1 and 3.2 provide no focus on test model abstraction levels. Several other studies presented in Sections 3.1 and 3.2 focus on lowering the cost of testing by reducing the test suites, preserving the original coverage, though at the expense of fault-detection ability. Like [29], we investigate the fault-detection effectiveness of reduced test suites, yet based on a different idea. Whereas test-reduction techniques are based on removing tests in a test suite that do not contribute in increasing the fault-detection ability, this thesis rather focuses on abstracting the test model itself. This means that not only are the number of test cases in the test suites reduced, but also the detail level in the test model.

Last but not least, as sneak-path testing has appeared to be of high importance in software testing, this thesis also seeks to provide more empirical data on this particular type of testing.

To conclude, existing research have evaluated state-based coverage criteria. As the attention has mostly been directed towards fault-detection effectiveness, there is still a lack of empirical results regarding the cost of such testing. Especially, regarding how state-based test criteria perform when being exposed to real faults. The majority of the results are based on studies where artificial mutation operators have been applied in small academic programs. The few studies that do use (or partly use) real faults [15, 16]) tend not to describe how those faults were collected.

In summary, this thesis complements and extends existing research on the cost-effectiveness of SBT by

- **using** an industrial safety-critical system,
- **using** real faults (from an industrial field study) in mutation analysis,
- **comparing** six state-based coverage criteria,
- **performing** a comparison of two test oracles,
- **studying** the impact of varying the test model abstraction level, and
- **applying** sneak-path testing.

The next part of this thesis presents the development of an extensible model-based testing tool.

PART II – DEVELOPMENT OF AN EXTENSIBLE MODEL-BASED TESTING TOOL

Part II regards the first part of the research goal (introduced in Section 1.2), and presents an extensible model-based testing tool. Chapter 5 addresses the design and the development of the tool, whereas Chapter 6 discusses experiences from the development process.

5 Development of a Model-Based Testing Tool

Motivated in Chapter 1 by the lack of extensible and configurable model-based testing tools, this section proposes a MBT tool, TRansformation-based tool for Uml-baSed Testing (TRUST), whose software architecture and implementation strategy facilitate its customization to different contexts by supporting configurable and extensible features such as input models, test models, coverage criteria, strategies for test-data generation, and test-scripting languages.

In this thesis, *configurability* is defined as the ability of selecting between several options, provided by the tool, for a specific feature. For example, the tool is configurable with respect to coverage criteria if it lets the user select among several coverage criteria such as all transitions and all round-trip path coverage criteria [31]. *Extensibility* is defined as the ability of providing more options for a feature without any modification in the components that are not responsible for the feature. For example, providing support for generating test scripts in more languages is considered as extending the tool.

The approach presented, which is inspired from the Model-Driven Architecture (MDA) standard [104], relies on a series of model transformations to generate test cases. The main idea is to design a tool in such a way that its different components provide and require standard interfaces with input and output models based on standard metamodels. Each component in this tool is responsible for one feature (e.g., test model, test data, etc.) involved in the process of generating test cases. This separation of concerns and provision of standard interfaces make TRUST configurable and extensible. In addition, model transformation technology helps the developer upgrade the components with a new set of transformations from standard inputs into well-defined outputs.

The approach allows instantiating new, context specific MBT tools by extending or configuring TRUST with customized features, such as input models, test models, coverage criteria, strategies for test-data generation, and test-scripting languages.

The remainder of this chapter describes in detail the model-based approach to develop the automated tool TRUST.

5.1 Requirements, Design, and Implementation of TRUST

This section discusses the main requirements of TRUST, define and justify architectural decisions, and choices of technologies. In addition, details regarding the test-case generation and test-case execution procedure will be provided.

5.1.1 Requirements and Approach

To be optimal from a practical standpoint, a tool that supports all features of UML is desired. However, as discussed in Chapter 3, this may not be enough – extensibility to various UML extension profiles, such as MARTE [95] and UML QoS profile [105], may also become a requirement in future applications. What is needed is a tool that shows versatility in various contexts. Adding different output scripting languages (such as C++, Python, and Java), test models (such as transition trees and testing-flow graph [106]), coverage criteria on test models (such as all transition and all round-trip path coverage), and test-data generation techniques (such as random and adaptive-random search [107]) for different application domains and systems are examples of useful extensions for TRUST.

In addition, the tool should be easily configurable to satisfy varying requirements, which means that the user should be able to easily configure features such as input model, coverage criteria and strategies for test-data generation. High configurability enables testers to experiment with various techniques without significant effort and changes in the tool implementation. This is of practical importance as different test models, coverage criteria, and test-data generation techniques helps in targeting different types of faults. Figure 1 shows a summary of the tool's high-level requirements.

The approach taken for implementing TRUST (Figure 2) is based on model transformations. The idea (inspired by MDA concepts, introduced in Chapter 2) is to generate a test model using a series of horizontal (endogenous and exogenous) model-to-model transformations on an input design model, modeling the platform independent model (PIM) of a system. Then, a vertical, exogenous model-to-text transformation is used to generate test scripts.

This approach was found to be very well-suited for developing TRUST, firstly due to the fact that it addresses the stated extensibility and configurability requirements. Each component of TRUST implements one set of transformation rules (e.g., transformation from test tree to test cases). Each component has well-defined interfaces with other components. More specifically, each interface provides output to, or requires inputs from, other components by means of intermediate models conforming to metamodels. Secondly, separation of concerns among components has made each transformation responsible for providing one feature such as test model, test data, and test scripts. Therefore, adding a new feature (for example, output test scripts in a new language) can be achieved by writing a new set of transformation rules in one of the components, without affecting the other components. Thirdly, in the model-transformation based approach, the transformation language provides the developer with direct support for navigating, creating, and manipulating a model, based on its metamodel. Generally, the transformation rules are relatively compact and easy to read, write, and change.

REQ1.	The tool should handle UML diagrams such as state machines, sequence diagrams, activity diagrams, and OCL as the constraint language for UML diagrams
REQ2.	The tool should have a configurable and extensible input model
	REQ2.1. The tool should be extensible for new UML profiles and configurable for existing UML profiles
	REQ2.2. The tool should be extensible for changes in the UML metamodel and configurable for existing UML metamodels
REQ3.	The tool should have a configurable and extensible testing strategy
	REQ3.1. The tool should be extensible for new test models and criteria and configurable for existing ones
	REQ3.2. The tool should be extensible for new test-data generation strategies and configurable for exiting strategies
REQ4.	The tool should have a configurable and extensible test-scripting output language
REQ5.	Extending the tool should be as easy as possible
	REQ5.1. Making any extension for the tool should have minimum effect on the tool's architecture
	REQ5.2. Extending or changing one component of the tool should be possible using knowledge of its interfaces and should not require any knowledge about inner details of other components of the tool

Figure 1 High-level requirements of TRUST

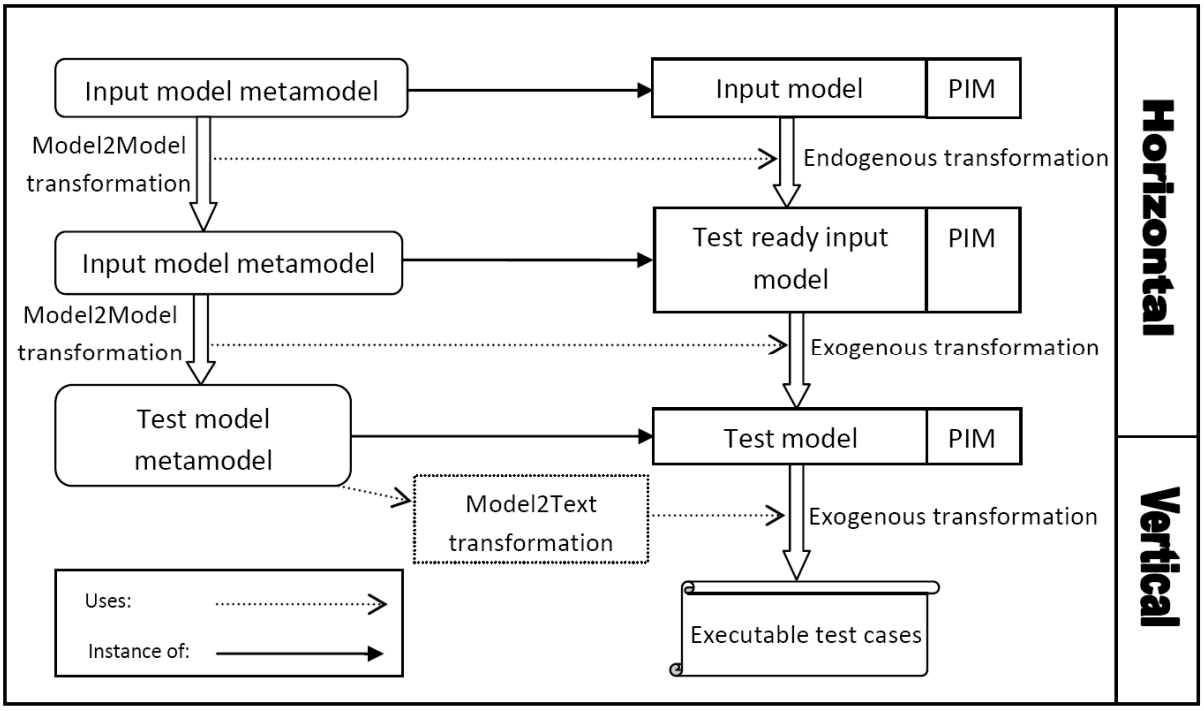


Figure 2 Model-transformation based approach for test-case generation

In this thesis, TRUST was configured with UML 2.0 state machines as the input model. REQ1 in Figure 1 is refined accordingly as follows: The tool should accept UML 2.0 state machines with support for concurrency and hierarchy. Constraints on state machines may be written in OCL because it is an OMG standard for writing constraints on UML diagrams. Furthermore, the general model-transformation based approach, given in Figure 2, is instantiated on UML 2.0 state machines, as shown in Figure 3.

Required activities, technologies, and the procedure for this approach will be explained in the remainder of this section.

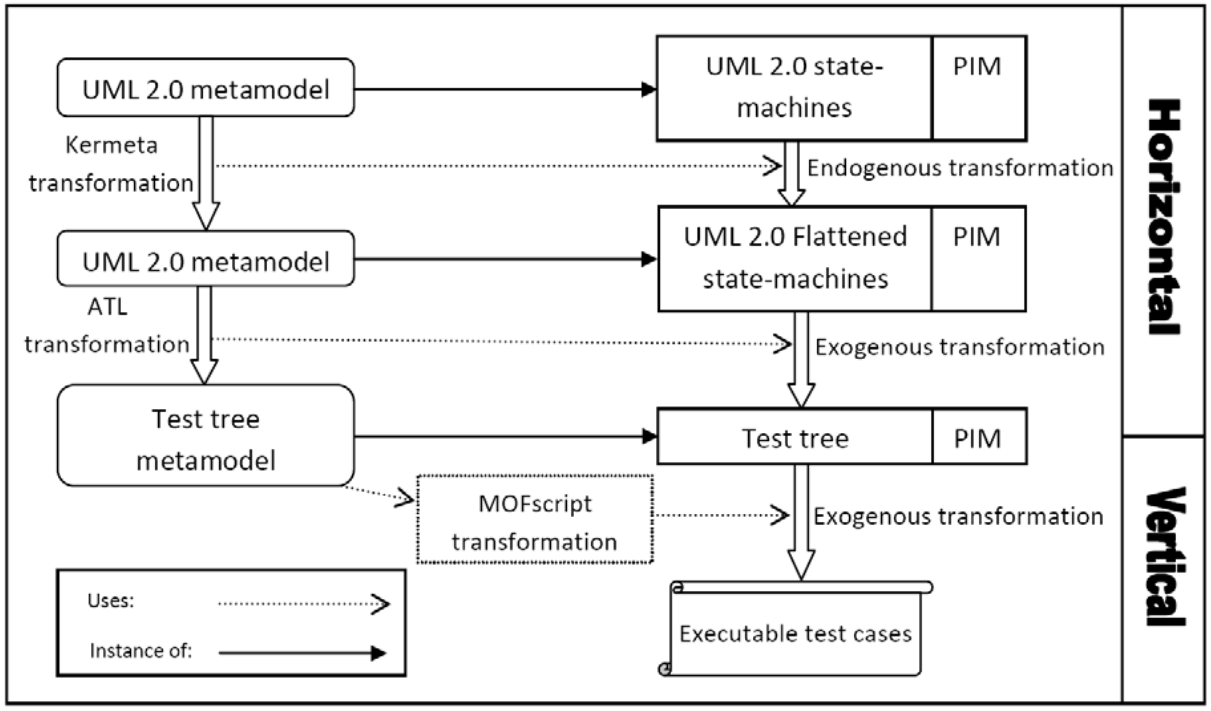


Figure 3 Model-transformation based approach for TRUST when configured for state machines

5.1.2 Development of TRUST using a Model-Transformation Approach

This section describes the activities involved in developing TRUST when configured for state-based testing (SBT). Firstly, transformations must be specified and implemented for various activities (activities A1, A3, and A6 in Figure 4). Since these transformations are applied on metamodels, we may need to define input metamodels (activity A2 in Figure 4). However, since some metamodels already exist (e.g., the UML 2.0 metamodel), it is not necessary to define all metamodels from scratch. Secondly, to make the test cases executable, test data is required (activity A5 in Figure 4). Finally, a method for evaluating the OCL constraints that are defined on UML state machines must be developed (activity A4 in Figure 4). In Figure 4, the notes associated with the activities show the technologies selected to implement each activity. These choices will be justified below.

From UML State Machines to Test Models

The first SBT activity is flattening the input state machine. As explained, the input behavioral model is a UML 2.0 state machine that allows complex structures like simple composite states, orthogonal states, and submachine states. Testing can be performed directly on such state machines, but this requires rather complex strategies, because such structures complicate the traversal and analysis of the state machine. An alternate approach is to flatten the state machines first, by removing concurrency and hierarchy, and then apply a testing strategy. In

order to obtain a better separation of concerns and lesser analysis complexity, the latter was implemented.

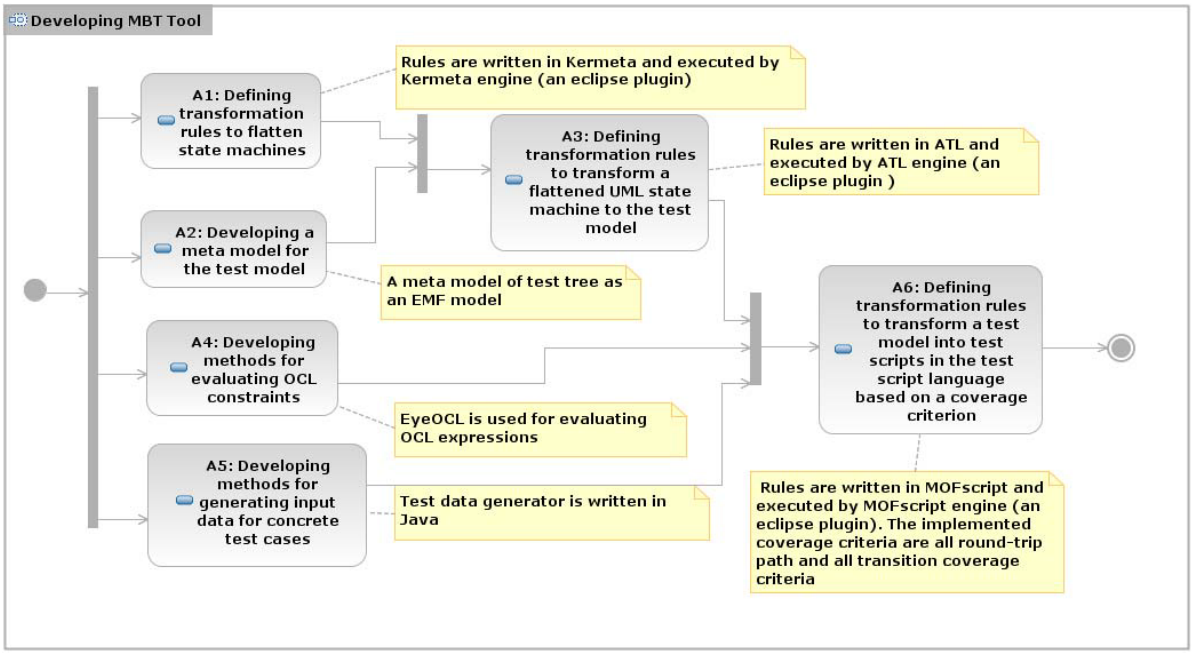


Figure 4 Activities and technologies for developing TRUST using model transformations

Several algorithms are reported in the literature to flatten concurrent and hierarchical state machines [31, 108]. However, to the author's knowledge, these algorithms are partial and do not provide flattening of both hierarchy and concurrency. Therefore, we decided to implement a self-written flattening algorithm for UML 2.0 state machines. The implemented algorithm is a stepwise process that allows the user to modify the UML model at several points during the transformation towards the flattened version. The first step in the flattening process is to search all nested levels for submachine states and transform these into a set of simple composite states. Next, all simple composite states with one region are transformed to a set of simple states or orthogonal states. If there are orthogonal states present in the model, these may now be transformed to simple composite states. Finally, the simple composite state(s) created in the previous step are transformed to a set of simple states.

The result is a state machine consisting of an initial state, simple state(s) and possibly a final state. The flattening follows a set of transformation rules implemented in Kermeta [109]. The key aspects in these rules address (1) how to combine concurrent states, and (2) how to redirect transitions. Redirecting transitions may require duplicating transitions, changing source or target states, and combining transition information (triggers, guards, transition activities, and state entry/exit activities). Interested readers may consult [34] for more detailed

information about the flattening algorithm and its corresponding transformations and implementation.

Figure 5 shows a Kermeta rule used for flattening of simple composite states. Incoming transitions to an entry point in a simple composite state are redirected to each of the outgoing transitions of the same entry point. The small example below provides the Kermeta rule that identifies the incoming transitions that will be redirected by calling another Kermeta rule.

Once the flattened state machine is generated, it is transformed into a test model. This transformation requires three inputs: the source metamodel, the source model (which is an instance of the source metamodel), and the target metamodel. The source metamodel is a metamodel for the flattened state machine, which is the same as the target metamodel for the flattening-transformation step in Kermeta. The output of the Kermeta transformation, a flattened state machine, provides the second input, which is the source model. The last input, the target metamodel, is a metamodel for a test model. The expected structure and content of a test model is strongly dependent on the selected testing strategy. In the current version of TRUST, the test model conforms to a test-tree metamodel. Figure 6 shows the metamodel developed as an Ecore file, based on EMF (Eclipse Modeling Framework). The metamodel represents a tree with a starting node, called alpha, and its outgoing edges. Each edge in the tree has a target node and may have several children that are the target node's outgoing edges. Similar to transitions in a UML state machine, each edge in the test-tree metamodel is associated with at least one trigger and may have an associated guard and effect. In addition, each node may have an associated state invariant.

```
/**
* Rule getTransitionsTargetedInEntryPoint identifies the incoming transitions to this
* vertex which is an entry point.
*/
operation getTransitionsTargetedInEntryPoint (r: Region) : Set<Transition> is do

    //create a set of all incoming transitions to this entry point
    var IN : Set<Transition> init Set<Transition>.new

    //add each incoming transition to the set
    r.transition.each{t |
        if (t.target.isInstanceOf(Pseudostate))
            then if (t.target.asType(Pseudostate).kind == PseudostateKind.entryPoint)
                then if (t.target.asType(Pseudostate) == self)
                    then IN.add(t)
                end
            end
        end }

        //return transition set
        result := IN
```

Figure 5 Example of Kermeta flattening rule

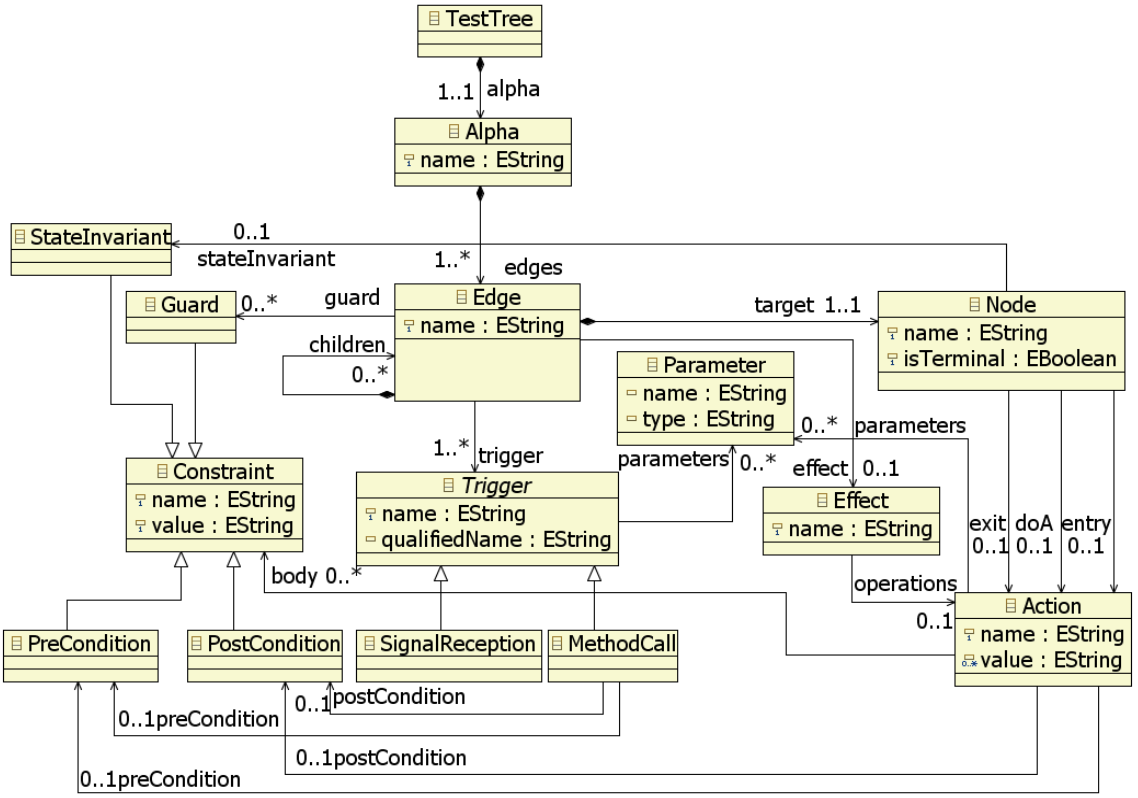


Figure 6 Test-tree metamodel for the EMF

The flattened state machine is transformed into a test tree (e.g., a transition tree for all round-trip paths coverage criterion) by a set of ATL transformation rules that take the three abovementioned inputs as parameters. We chose the ATL transformation language because most mappings in this step are simple (mainly one-to-one), which makes a declarative approach the best choice. In declarative approaches, as opposed to imperative ones, control flow and the application order of rules are not explicit.

Usually the transformations in declarative languages have less transformation code and are more comprehensible than imperative languages [110]. This transformation could also have been implemented with any other declarative, hybrid, or even imperative language. In this research, however, it was decided to stick with an Eclipse-based technology so as to develop the entire tool on a consistent platform. For example, the transformation rules, instantiating a transition tree from the test-tree metamodel, start from the initial state of the state machine which is mapped to the root node, called the Alpha node, in the test tree. All outgoing transitions of the initial state are also mapped to the outgoing edges from the Alpha node.

This process of mapping transitions to edges is applied recursively on the target state of all transitions in the state machine. Finally, the recursive rule stops when it reaches a leaf or a state that has already been visited in the same path starting from the Alpha node (all round-trip paths coverage [111]). An example of ATL transformation rules is shown in Figure 7 that maps the Constraint metaclass (`SM!Constraint`) including its name and value properties from UML metamodel (SM) to the Constraint class (`transitiontree!Constraint`) in the test tree metamodel (transitiontree) and its name and value properties.

```
rule Constraint2Constraint{
    from
        a:SM!Constraint
    to
        b:transitiontree!Constraint(
        name <- a.name,
        value <- a.specification.stringValue()
        )
}
```

Figure 7 Example of transformation rule implemented in ATL

From Test Model to Executable Test Cases

The generated test tree is the input for the next transformation, which generates the executable test cases. The MOFScript [112] transformation language was chosen for several reasons. Firstly, it supports the MOF standard [113], which means that it can transform any MOF-based model-to-text. Secondly, it is an imperative language for writing transformation rules similar to many programming and scripting languages. This makes the MOFScript language easy to use and understand. Thirdly, MOFScript provides access to external Java libraries. This makes the language very suitable for the context of this thesis because of the need to access a test-data generator (implemented in Java) during the transformation to obtain test data. MOFScript transformations require the source model and its metamodel, which are readily available from the previous step. There is no need to provide the grammar of the output language as an input to TRUST, but of course defining transformations requires its definition.

Each path in the test tree represents an abstract test case. Thus, an abstract test case consists of a sequence of nodes and edges. Nodes are mapped from states in the state machine and states are defined by state invariants, which are OCL constraints serving as test oracles. An edge contains all the information related to the trigger including event (e.g., an operation call or a signal reception), a guard, and an effect from the state machine's transitions.

The MOFScript transformation traverses the test tree (e.g., the transition tree) to obtain the abstract test cases and transforms them to concrete (executable) test cases, which are written in a test-scripting language. However, it is possible to generate several concrete test cases from an abstract test case by using different test-data values. There are many possible approaches for generation of test-data [114, 115], which are applicable in different situations. What was implemented in the first version of TRUST is the simplest method, which is random data generation for operation calls. This test-data generator is written in Java and provides random values for the parameters of triggers. However, such a test-data generation technique is not suitable when transitions are guarded and parameters of the triggers are used in the guards. The MOFScript rule shown in Figure 8 illustrates how a trigger is mapped from the test tree for all transitions to C++ test cases. The rule maps the name of the trigger event operation and then uses another mapping rule `mapParameter` to map the parameters for the trigger event operation.

```
/**
* rule 'mapTrigger' generates the C++ code to invoke the operation implementing
* the trigger event. Rule 'mapParameter()' is called to map parameters in the
* trigger event operation call. Each trigger is either a MethodCall,
* SignalReception, or Timer.
* @param triggerWithParam List, the total generated output for a trigger as String.
* @param noOfParam Integer, temporary helper variable used for counting parameters.
*/
transitiontree.Trigger::mapTrigger(){

    var triggerWithParam : List;
    var noOfParam : Integer = 0;
    if(self.oclGetType().equals("MethodCall") or self.oclGetType().equals("SignalReception") or
    self.oclGetType().equals("Timer"))
        {
        triggerWithParam = controlClassReference+"->"+self.name + "(";

        if(not self.parameters.isEmpty()){
            self.parameters->forEach(p:transitiontree.Parameter){
                triggerWithParam += p.mapParameter();
                noOfParam = noOfParam + 1;

                if(noOfParam < self.parameters.size()){
                    triggerWithParam += ", ";
                }
            }
        }
        triggerWithParam += ");";
    }
    return triggerWithParam;
}
```

Figure 8 Example of transformation rule implemented using MOF Script

When executing test cases, OCL expressions in guarded transitions should be evaluated at runtime to detect failures. For the same reason, the state invariants associated with states must also be evaluated at runtime. One way to evaluate such OCL constraints is to translate them into a test-scripting language. The constraints will then be evaluated during the execution of the test scripts. Compiler technologies [116, 117] may be used to translate constraints in one language to constraints in another language. This approach however is not reusable across contexts with different test-scripting languages. For example, if we have transformation rules that transform OCL constraints to C++ and the test-scripting language changes to Java, it is then required to define new transformation rules from OCL constraints to Java expressions. An alternate approach is to use an existing OCL evaluator [52, 54, 55, 53] that is called during the execution of a test case to evaluate the OCL constraints. This approach requires an object model of the SUT at runtime, representing the current state of the system. This model along

with the constraint to be evaluated is passed to the OCL evaluator, which in turn returns the result of the evaluation. This approach is reusable across contexts because the only required change for each output language is to use its appropriate invocation method for calling an external library (the OCL evaluator). On the other hand, this approach may slow down the test execution as the OCL evaluators are being called at runtime. In both approaches, we need a mechanism to query the current state of the SUT and evaluate constraints on the current state of the SUT. Querying the current state of the system depends on the implementation of the SUT and the test-scripting language's facility to access the state of the SUT. For instance, if the SUT is implemented in C++ and test-scripting language is C++, the state of the SUT may be queried using getter methods of the SUT.

Since the tool was preferred to be reusable in different contexts, it was decided to use an OCL evaluator that can be invoked from test scripts. Therefore, an efficient evaluator, in terms of evaluating expressions, was chosen that, for example does not require to be called several times for evaluating a single expression. After investigating several OCL evaluators such as OCLE 2.0 [52], OSLO [53], IBM OCL parser [54], and EyeOCL Software (EOS) evaluator [55], EOS was chosen as it proved to be the most useful evaluator for the stated requirements (see Figure 1). Since EOS is a Java package, to invoke methods from its classes we need to have access to Java from a test script. Java Native Interface [118] was used to access the EOS in test scripts in C++.

To evaluate OCL expressions, EOS requires class and object diagrams to be loaded into its memory. In order to accomplish this, another MOFScript transformation was written, which takes the UML class diagram (modeling state variables, method calls, and signal receptions of the SUT) as input and generates a Java wrapper class that includes a set of EOS method calls for making class and object diagrams. This wrapper class is generated before executing the transformation of the test tree to test scripts. During test executions, the required object model is created based on the current values of system state variables.

5.1.3 Generation of Test Cases

This section discusses how the activities introduced in Section 5.1.2 was designed and implemented. Figure 9 depicts the architecture of TRUST, which consists of five components. Table 4 shows the mapping between each activity and a component. Each component has provided and required interfaces with other components to ease extensibility and configurability, as discussed in Section 5.1.1. Each interface provides or requires models that are instances of well-defined metamodels. For example, the *TestModelGenerator* component in Figure 9 requires an interface from the *TestreadyModelMaker* component to access the flattened state machine, which is an instance of the UML metamodel. In addition, the *TestModelGenerator* component provides an interface to the *TestScriptGenerator* component to access the test tree (e.g., the transition tree), which is an instance of the test tree metamodel. The architecture shown in Figure 9 was developed with the aim to support extensibility and configurability. For instance, if TRUST needs to be extended to handle C++ as an output test-scripting language, the only component to modify is the *TestScriptGenerator* component where new transformation rules in MOFScript must be defined. The other components do not require any change. Each component also has clearly defined configuration parameters that can easily be adjusted. For instance, if the coverage criterion is to be changed from all round-trip paths to all transitions, we only need to change the input coverage criterion in the *TestScriptGenerator* component.

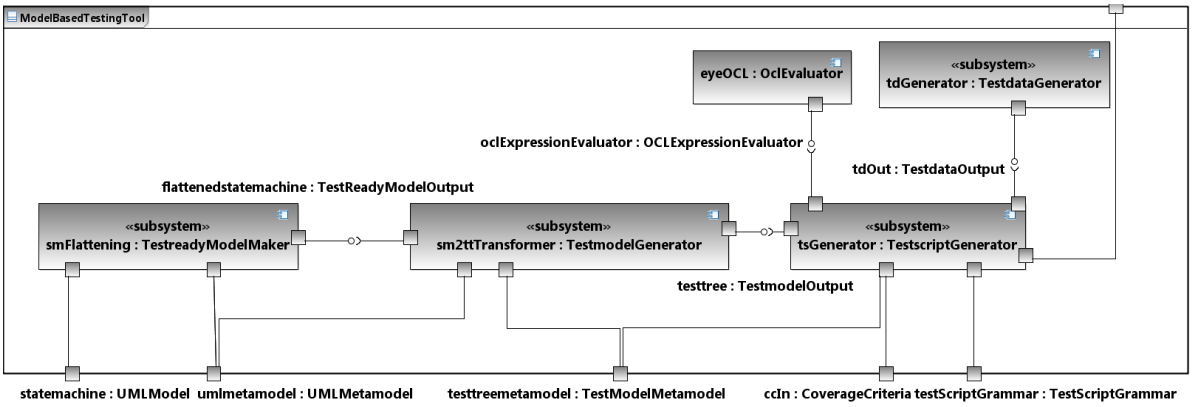


Figure 9 Architecture diagram of TRUST configured for UML state machines

Table 4 Activities implemented by each component

Component	Activity
TestreadyModelMaker	A1
TestModelGenerator	A5
TestScriptGenerator	A6
OCLEvaluator	A3
TestDataGenerator	A4

Figure 10 shows interactions between different components that take place at runtime when TRUST is executed with an input state machine. The state machine is passed to the *TestreadyModelMaker* component, which flattens the state machine and passes the flattened state machine to the *TestModelGenerator* component. This component generates the test model from the flattened state machine and passes it to the *TestScriptGenerator* component. The *TestScriptGenerator* component determines if a trigger (a method call or a signal reception) in a test script needs static test data. Test data can be generated statically if values can be determined prior to execution of the test script, or dynamically in the other case. The parameters whose values can be determined only at runtime are obtained at this point. Section 5.1.4 provides more specific examples of static and dynamic test-data generation.

In the initial version of TRUST, static data for a parameter is generated randomly from the possible set of values. Generating random test data may not be appropriate when the parameter of a trigger is used in the associated guard. In this case, a parameter value that satisfies the guard must be chosen, so that the trigger can be fired. However, if the value of a parameter is selected randomly from a large set of possible parameter values, then the possibility of selecting the parameter value that satisfies the guard may be very low.

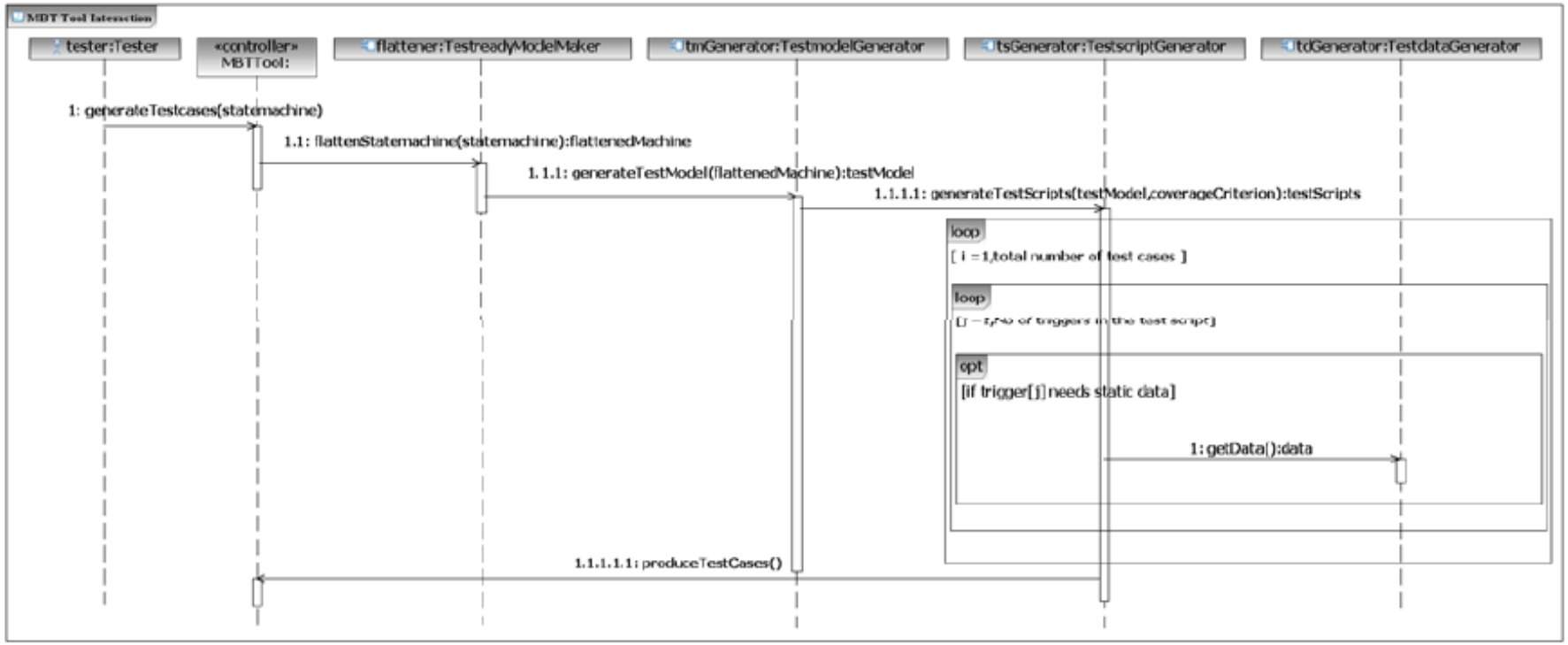


Figure 10 Interactions between different components of TRUST

5.1.4 Execution of Test Cases

Figure 11 shows how a test driver interacts with the SUT and other external tools when it executes a test case. Each test case consists of a series of triggers (methods or signals) with optional guards. The state of the system is checked before and after executing each trigger based on state invariants written as OCL constraints. When a test case is executed, the test driver initializes *EyeOCLWrapper* and *TestDataGenerator* (messages 1.1 and 1.2, respectively). Afterwards, the test driver obtains the state of the SUT by interacting with the SUT (message 1 in the *loop* fragment). The state of the SUT can be obtained in many ways. If the implementation of the SUT is in an object-oriented language such as in Java or C++, the state variables can be accessed using getter methods of classes if they are available. Alternatively, if the source code is available, but there are no getter methods, the source code needs to be instrumented before and after each method call to obtain the current state of the SUT.

In the other case, if the source code is not available, then there are two possible options. The first option is that the system state might be obtained using some facilities of the implementation language. For example, if the Java byte code of the SUT is available, then Java's reflection facility [119] can be used to access the system state. The second option is to use a test-scripting language that provides some mechanisms to access the state of the SUT..

In the implementation of TRUST that was used in this thesis, the test driver retrieves the minimum system state information by querying the values of only those state variables that are used in the OCL constraint which is to be evaluated, in addition to querying the state pointer implemented as an array in the control class of the SUT holding the pointer to the current state object. After obtaining the current state of the system, the test driver creates an object diagram using *OCLEvaluator* via the *EyeOCLWrapper* class (message 2 in the *loop* fragment). This class automatically generates an implementation of the class diagram and instantiates the object diagram corresponding to the implementation at runtime, based on the current state of the system (message 2.1 and 2.2 in the *loop* fragment). The test driver then evaluates the expected system state using *OCLEvaluator* via the *EyeOCLWrapper* class (message 3 and 3.1 in the *loop* fragment). Once the system state is evaluated against the expected state, the trigger, which may be guarded, should be executed. If dynamic test data is required for the trigger, the test driver communicates with the *TestDataGenerator* class (message 1 in the *opt* fragment) to obtain required values. Whether the value for a parameter must be generated at runtime is indicated in the data model of the SUT. During the test case

generation, TRUST checks if a parameter requires dynamic data generation or if static data is readily available.

In the case of guarded triggers, the associated guard must be evaluated before executing the trigger and after obtaining the dynamic test data. The guard may contain system variables and input parameters of the trigger. This means that in order to evaluate the guard, we need to obtain the system state and the values of the parameters (possibly dynamically generated) involved in the guards at runtime; the static and dynamic parameters that are used in the guard are replaced with their current values obtained from *TestDataGenerator* dynamically or statically. The guard is then evaluated in the same way as the state of the system was evaluated (messages 4, 5, 5.1, 5.2, 6, 6.1). Once the guard is evaluated, the appropriate method (i.e., the method that implements the trigger event) is invoked on (or signal is sent to) the SUT (message 7). After the execution of the method (or reception of the signal), the state of the system is evaluated (message 8, 9, 9.1, 9.2, 10, 10.1) in the same way as the previous state and guard evaluations. This process is repeated for all triggers in the test case. Finally, cleanup operations are performed on the SUT (message 1.3) once all the triggers have been executed on the system. These operations release the resources used by a test case such as memory and CPU.

The next chapter presents experiences and lessons learned from the development of TRUST.

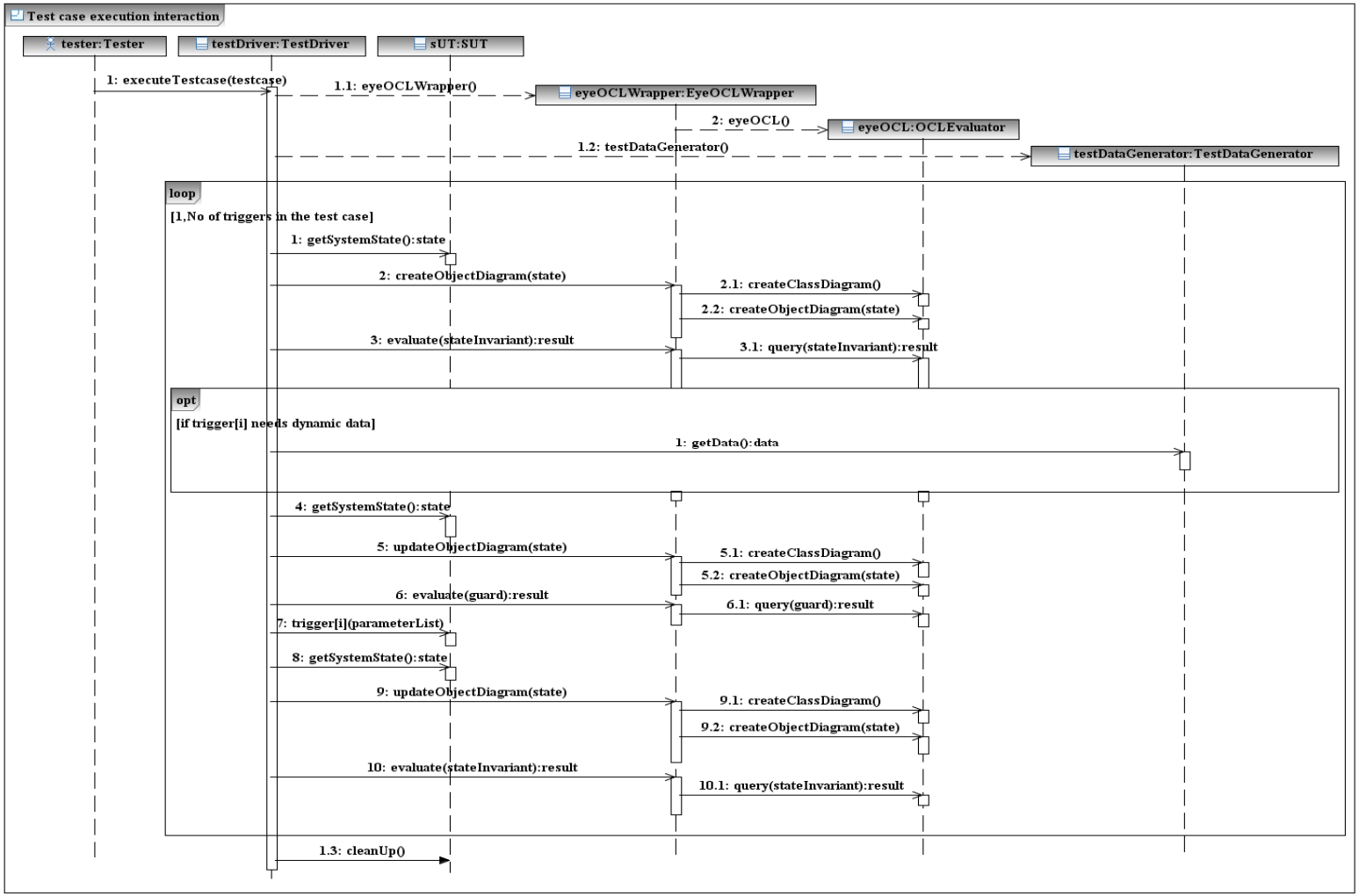


Figure 11 Interactions between different components while executing a test case

6 Lessons Learned

Developing TRUST and then applying it to real world case studies taught us some important lessons about both modeling and model transformations. In this section, we will discuss the lessons learned for these aspects.

6.1 Modeling the SUT

In this thesis, precise behavioral modeling of complex industrial systems using standard UML 2.0 state machines was a prerequisite for using TRUST. The flattening component requires a correctly specified state machine; currently, no feedback is provided in case of errors in the model. Modeling correctly, however, is not a trivial task and requires careful studies of the UML specification. Even though constructs like concurrency and hierarchy are supposed to ease the understandability of large state machines, such constructs may actually confuse the developer. In particular, we experienced that concurrency, if not carefully applied, could introduce modeling errors in practice. For example, concurrent regions sometimes make it difficult to see the set of transitions between state combinations. A typical fault could be that a guard is missing on a transition, which allows for transitions to state combinations that are illegal targets from particular source states. However, we found that it helped to inspect the flattened state machine to detect such mistakes. During inspection, it was detected that a missing guard on a transition from an initialization state to a system running state in Region 1 would allow transitions to be incorrectly fired in Region 2.

Lesson learned: Careful inspections of the interaction between concurrent regions are necessary to prevent unintended behavior. Moreover, constructs like concurrency and hierarchy must be carefully inspected to ensure that they are used correctly.

6.2 Model-to-Model Transformation Technologies

The model-to-model transformations in TRUST used two different transformation languages: Kermeta and ATL. Kermeta appeared to be highly appropriate for flattening UML state machines. In addition to being an object-oriented language, it allows you to add behavior to the metamodel through aspect weaving. However, we experienced that navigating in the metamodel was rather time consuming. Alphabetically organized in a super-sub class structure, the UML 2.0 metamodel is a complex model that is difficult to navigate. Having

tool support integrated in the Kermeta plug-in that could remove abstract classes and instead present the concrete classes relevant for a particular purpose would have been very useful.

Since the metamodel for test trees is relatively simple, the transformation from the flattened state machine to the test model was expected to be straightforward and easy to implement by depth-first traversal of the state machine using a declarative language (ATL). However, we found that the declarative programming style was not intuitive to handle, perhaps because most developers are used to imperative programming languages. Even though the final ATL code for test model generation is very short, debugging it was quite difficult especially when the input model was big. As the input state machine was quite large it caused Eclipse to run out of memory while generating a transition tree for all round-trip paths coverage criteria. This was due to the many recursive rule calls required to generate the transition tree from the flattened state machine. Implementing transition tree generation using recursion was the only possible option when writing rules in the ATL language in a declarative fashion. Technology-wise, we also faced many problems while debugging the ATL rules, especially when the input models are large causing the debugging interface to hang.

Lesson learned: In future expansions of TRUST, alternatives to ATL used for transforming state machines to test trees should be explored and considered.

6.3 Model-to-Text Transformation Technology

Developing the final set of transformations in MOFScript was the easiest part of developing TRUST, because the rules are defined in an imperative form. MOFScript is quite similar to programming languages like Java, and provides powerful features that are easy to use for querying models, outputting text, and accessing external Java libraries. We did not face any special challenges while using MOFScript for generating test scripts.

Lesson learned: MOFScript is a favorable model-to-text technology in terms of ease of use.

PART III – COST-EFFECTIVENESS ANALYSIS

Introduction

We will now address part (2) of the research goal, introduced in Section 1.2, by presenting a cost-effectiveness analysis of

- six state-based coverage criteria,
- using two different oracles,
- using two test models with two different level of details, and
- sneak path-testing.

Chapter 7 describes the design of four case studies that were conducted. Chapter 8 presents results from the first case study that addresses cost-effectiveness analysis of the coverage criteria all transitions (AT), all round-trip paths (RTP), all transition pairs (ATP), paths of length 2 (LN2), paths of length 3 (LN3), and paths of length 4 (LN4). The second case study (Chapter 9) raises the question: How does varying the oracle affect the cost-effectiveness? The influence of the test model abstraction level on the cost-effectiveness is subject of Chapter 10. Finally, Chapter 11 studies the impact of sneak-path testing on the cost-effectiveness.

7 Design of Case Studies on the Cost-Effectiveness of State-Based Testing

Section 7.1 gives the rationale for applying the case study research method for the purpose of evaluating the cost-effectiveness of SBT. Section 7.2 describes the study in detail.

7.1 Rationale for Selected Research Method

In general, the choice of research method depends on (i) the prerequisites for the investigation, (ii) the purpose of it, (iii) available resources, and (iv) how the collected data is preferred to be analyzed [59]. A prerequisite for this investigation in particular, as being part of and mainly funded by the EVISOFT project, was to conduct research aiming at empirically evaluating model-based development using UML state machines at ABB. This implies application of the technique within an ABB context. A prerequisite for state-based testing is that of state-based modeling. State-based development is a contemporary phenomenon that should be studied within its real-life context; it is difficult to separate from its context as its use depends on the type of system, the type of maintenance task, the knowledge of the developers, and the tools they use. As what comes to resources, the degree of ABB involvement during the project expiration varied due to limitations in available resources. At the time the project started, four ABB researchers and developers participated in design and implementation of the system ABB provided to the EVISOFT project. Also, ABB was highly involved in recruiting subjects for the purpose of collecting fault-data. ABB was not involved, however, in the development of the model-based tool for making such state-based testing feasible, neither in the evaluation of state-based testing. This illustrates the combination of involved actors. Finally, both quantitative and qualitative analyses are desired for exploring the research objective.

The preceding paragraph points in the direction of a rather complex investigation; different types of data collection from observations of several perspectives are required. The case-study method was decided to be a suitable approach due to state-based testing being a contemporary phenomenon that should be studied within its context. Furthermore, because it is a complex topic, which is difficult to explore purely by quantitative data, the techniques used in case studies are relevant for collecting the desired qualitative data. Also, the possibilities for triangulation of data will help to reduce misinterpretations of study results. Triangulation may be used to collect data from several sources and hence create data redundancy – the case may then be seen from different angles: “Triangulation has been generally considered a process of using multiple perceptions to clarify meaning, verifying the

repeatability of an observation or interpretation" [120, p. 454]. In [121], a case study evaluation exercise is defined to be one where a method or tool is tried out on a real project. Furthermore, Wohlin *et al.* [59, p. 12] adds that within software engineering, case studies should *not only* be used to evaluate how or why certain phenomena occur, but also to evaluate the differences between, for example, two design methods. Hence, case studies are also appropriate when it comes to comparisons of technologies in order to find the best technology. The case study method is appropriate for these kinds of studies with respect to the type of research question posed, which is explorative; a comparison of different testing strategies at two test model abstraction levels with two oracles.

What is different from traditional case studies, though, is the extent of control the investigator has over actual behavioral events in parts of the study; in this study, the researcher had high control as regards to the testing phase.

7.2 Case Study Design

Four case studies were based on the same SUT, however addressing different research questions. Hence, this section describes what is common for the four case studies. Results from the case studies are organized in separate chapters (Chapters 8–11).

The study design describes how an evaluation of the cost-effectiveness of state-based testing (SBT) was carried out. The guidelines of Runeson and Höst [122] were followed for structuring the design of the study.

7.2.5 Research Objectives

This thesis presents an empirical study on automated SBT. The generated test suites were executed on a safety-monitoring component with state-based behaviour in a safety-critical control system developed in ABB's research department in Norway.

The main purpose of the investigation was to compare the cost-effectiveness of state-based testing using mutation analysis with real faults collected in a field experiment conducted in three global ABB departments.

Generating state-based test suites according to selected coverage criteria, however, is a highly demanding task, and not applicable as a manual process. Tools are required in order to make this a feasible approach in the industry. The model-based testing tool TRUST [33] was developed and used to generate the studied test suites (see Chapter 5 for details on requirements, design and implementation of TRUST). TRUST is a research prototype that

automates test-case generation according to a given test-scripting language, coverage criterion, and oracle through a series of model transformations.

The well-known coverage criteria (i) all transitions (AT), (ii) all round-trip paths (RTP), and (iii) all transition pairs (ATP) were chosen. The rationale for selecting these coverage criteria in particular, was that they are suggested by existing research to be representing weak to stronger fault-detection at different, yet reasonable, cost. In addition, test suites covering (iv) paths of length 2 (LN2), (v) paths of length 3 (LN3), and (vi) paths of length 4 (LN4) were also included in the study. Test suites are generated by traversing two, three, and four levels from the state machine's initial state. This is the most easy, uncomplicated approach to make a selection of test paths. Moreover, they represent a coverage that has not been included in related work. These three coverage criteria are the baseline used as comparison to the known AT, RTP, and ATP.

Two oracles of different precision levels were applied. Oracle O1 checks both the state invariant and the state pointer, whereas oracle O2 is restricted to only check the state pointer. The state pointer is a pointer to the current state object kept in an array in the control class of the SUT. In this particular case, the system was concurrent, thus keeping two pointers in the control class.

Moreover, to the author's knowledge there is a lack of observations on having the test model in different levels of detail. In this thesis, test suites were generated from two test models at different abstraction levels for the purpose of investigating the effect of increasing the test model abstraction level on the cost-effectiveness of the testing strategies.

Finally, the test suites were augmented with sneak-path testing to detect possible unspecified behaviour in the implementation.

It is expected that not all faults will be killed by every coverage criteria. Therefore, the reason for why faults are not being detected was examined in order to identify whether additional approaches could enhance the fault detection as diverse types of test techniques may target different types of faults.

The following research questions were addressed:

- RQ1: What is the cost-effectiveness of the state-based coverage criteria all transitions (AT), all round-trip paths (RTP), all transition pairs (ATP), paths of length 2 (LN2), paths of length 3 (LN3) and paths of length 4 (LN4)?
- RQ2: How does varying the oracle affect the cost-effectiveness?
- RQ3: What is the influence of the test model abstraction level on the cost-effectiveness?

- RQ4: What is the impact of sneak-path testing on the cost-effectiveness?

An overview of the research activities can be seen in Figure 12.

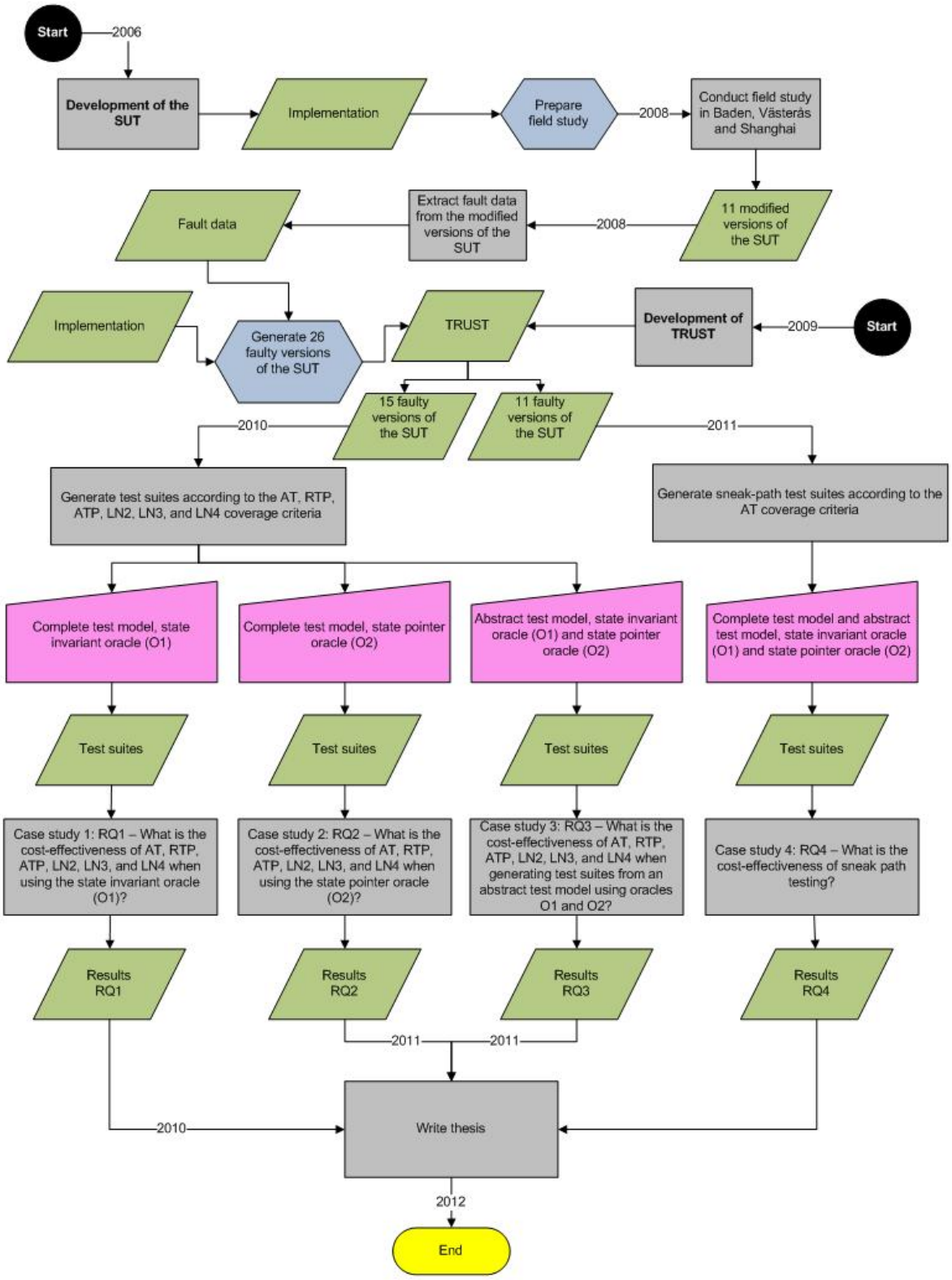


Figure 12 Overview research activities

7.2.6 Case Selection

The evaluation of the SBT was carried out in the context of a software process improvement project at ABB called EVISOFT.

7.2.6.1 Organization

ABB is a global company that operates in more than 100 countries with approximately 135,000[1] employees. A primary business area concerns power and automation technologies for which ABB develops software and hardware.

The ABB Corporate Research Center in Norway, NOCRC, is one of ABB's eight research centers worldwide. Their interests concern industrial communication, human-machine interaction, control and optimization of industrial processes, and instrumentation and operation of machines and processes. The research center supports the business units of ABB through projects and technical studies.

7.2.6.2 Software Process Improvement Project

ABB Corporate Research in Norway initiated a software process improvement initiative with the aim of improving UML modeling practices throughout the company.[2] This initiative was supported by the Norwegian Research Council through the project EVISOFT (EVidence based Improvement of SOFTware engineering), and conducted in cooperation with researchers from Simula Research Laboratory AS. The current project was initiated to investigate which UML diagrams may be beneficial to ABB in the process of going from the specification to the implementation phase, and for improving testing procedures. It provided a unique opportunity to assess the usage of precise, statechart-driven UML modeling and to evaluate state-based testing in a realistic safety-critical development environment.

A Technical Requirements Specification, developed by the business unit in cooperation with the scientists from NOCRC, was the starting point for developing a common understanding of the system. The modeling was a cooperative activity between Simula and NOCRC. Each modeling activity was closely monitored and followed by an introspective analysis of what happened. Based on these activities, several lessons are drawn regarding the benefits and challenges of precise UML modeling [14].

[1] From 2011.
[2] Previous results of the process improvement initiative work are reported in [1].

7.2.6.3 System Functionality for Selected Sub System

In order to satisfy safety standards (e.g. EN 954 and IEC 61508) and enhance the safety-critical behavior of their industrial machines, ABB developed a new version of a safety-critical system, the safety board, for supervising industrial machines. The safety board can safeguard up to six robots by itself and can be interconnected to many more via a programmable logic controller (PLC). It was implemented on a hardware redundant computer in order to achieve the required safety integrity level (SIL). The focus of this study is a subsystem of this system, called the *Application Logic Controller* (ALC). The main function of the ALC module is to supervise the status of all safety related components interacting with the machine, and to initiate a stop of the machine in a safe way when one of these components requests a stop or if a fault is detected. It shall also interface with an optional safety bus to enable a remote stop and a reliable exchange of system status information. The system will be configurable with respect to which safety buses the system can interact with.

The ALC sub system was chosen for this study as it exhibits a complex state-based behavior that can be modeled as a UML state machine. Complemented by constraints specifying state invariants, the state machine will be the main source in the process of deriving automated test oracles.

7.2.6.4 Modeling and Coding

The sub system was built as a joint effort between researchers from ABB SKCRC and Simula Research Laboratory. The ALC was developed according to a development method based on the use of UML state-machine diagrams for specifying a system's behavior. ALC was designed by using parallel and hierarchical UML state machines. It was implemented in C++ and the extended state-design pattern was used to ensure consistency between model and code. The first version of the system (including models and codes) was developed and verified by four developers (two coding, four designing) from the company and three researchers (one coding, three modeling).

Four initial meetings took place where the company representatives introduced the researchers to the domain and the functionality of the SUT to be developed. In addition to the company representatives, the initial requirements specification and design documents served as sources for identifying the system behavior. Throughout the meetings, the authors made initial versions of the state-based model of the SUT. One of the company representatives took an active part in the subsequent modeling iterations. As part of the modeling process, the requirements were discussed with several of the company representatives whenever questions

were raised and decisions had to be made. On many occasions, disagreements about the system specifications arose among stakeholders. In total, we spent approximately 320 person-hours on understanding (200 person-hours) and modeling the SUT (120 person-hours). It is important to note that, as opposed to cases where the system pre-existed the study, the modeling effort here could have been significantly smaller if the specifications had been stable to start with.

The SUT is described in a class diagram consisting of one control class, seven abstract classes, and 13 concrete classes. At the top most level, the resulting hierarchical state machine consists of one orthogonal state with two regions. Enclosed in the first region are two simple states, two simple composite states and 24 transitions. Each of the simple composite states contains two simple states. The second region encloses three simple composite states that again consist of, respectively, two, two, and three simple states. In addition, 18 transitions are present in the second region. This adds up to a maximum hierarchy level of two, 13 simple states and 42 simple transitions.

The state machines for the original version of the SUT can be found in Appendix C.

7.2.7 Data Collection Procedures

To measure the cost-effectiveness of SBT, four surrogate measures were used, inspired by the study of Briand *et al.* [12] and Briand and Labiche [123].

Cost is measured in terms of:

- Test-suite size – defined by the number of test cases in a combination of coverage criterion, test model, and oracle.
- Preparation time – number of seconds spent on preparing a test suite. The following actions are included: generating the test tree, and generating and building the test suite.
- Execution time – number of seconds spent on executing the test suite.

Effectiveness is measured by:

- Mutation score – defined by the number of faults killed by a test suite divided by the total number of seeded faults.

The remainder of this section addresses procedures for how the data collection was carried out. The main activities include (A1) preparation of test models, (A2) generation of mutant programs, (A3) extending and configuring the tool, (A4) generation of test suites, and (A5) execution of test suites on initial and mutated programs.

7.2.7.1 (A1) Preparing Test Models

The following sections address preparations of the original and the modified UML state-machine versions of the system introduced in Section 7.2.6.3 to be used as test models.

Original Complete Test Model

The original model, as first developed during the joint effort, was tested using TRUST and the RTP coverage criterion. However, some adjustments had to be done before being used as input to TRUST. There are multiple reasons for this.

Firstly, the implementation of the STATEMACHINEFLATTENER requires an initial state in the outermost level. Moreover, the flattening component does not support transitions that cross state borders. Hence, those transitions had to be remodeled to and from the super-state edges. This also implied adding initial states, entry points, and exit points in several of the super states. The STATEMACHINEFLATTENER neither supports multiple events on a single transition. Such transitions were thus resolved to one transition per event.

Secondly, a misinterpretation of the UML superstructure specification [124] caused a remodeling activity that involved replacing a choice point that was connected to an initial state with a simple state. Another choice-point case was resolved by replacing the initial state in a super state with an entry point. The explanation for this misinterpretation was the confusion of simple versus compound transitions. The UML specification states that the initial state can only have *one* outgoing transition. This, however, concerns the simple transition. Models having initial state directly connected to a choice point would, when flattened, be resolved to several outgoing transitions from initial state, which is not allowed. However, this is not the intended meaning of UML specification. The combination of initial state and choice points is allowed.

Table 5 Configuration parameters of TRUST for the ABB case

Parameter	Value
Input model	UML2.0 state machine
Test model	Test tree for all transitions
Coverage criterion	All round-trip paths
Test scripting language	C++
Test data generation technique	Random data generation
OCL Evaluator	EOS

TRUST was applied with the configuration values presented in Table 5 and the flattened state machine (see Appendix D) described in Table 6 as input.

Table 6 Features summary of the hierarchical scale of state machines

State machine feature	Unflattened	Flattened
Maximum level of hierarchy	2	-
Number of submachines	0	-
Number of simple composite states	5	-
Number of simple states	14	56
Number of orthogonal states	1	-
Number of transitions	53	391

After applying the flattening transformation and removing unreachable state combinations due to conflicting state invariants, the flattened state machine consisted of 56 states and 391 transitions, mostly guarded. In this case, TRUST was configured for the RTP criterion, applied on a test tree, which conforms to the same test-tree metamodel presented in [33]. The RTP criterion was chosen to provide an initial test run of TRUST.

Executing the generated test suites showed that a large number of test cases failed. Analyses of the execution results showed that this was explained by infeasible test cases as a result of (1) not having the correct values in the test data on guarded transitions, and (2) transitions between illegal state combinations in the flattened state machine. In order to address (1), the MOFScript transformation was modified to provide the required test data. Moreover, (2) was handled by removing illegal state combinations and the corresponding transitions from the flattened state machine.

Furthermore, even when a system is carefully designed and implemented, there is always a risk for introducing discrepancies between the specification and the implementation. Minor inconsistencies between the specification and the original version of the SUT were found during AT, RTP, and ATP testing – these inconsistencies, however, had to be resolved in order to run the tests without errors before moving on with the experimentation. Examples of such inconsistencies are mismatches between the specification and the implementation related to parameters in operations. This caused several test cases to fail, as there were no matching operations in the SUT. Such inconsistencies had to be handled in order to run the tests without errors and before moving on with the experimentation.

Modified Complete Test Model

The original version of the system was then modified by the author to include a fourth mode – the `ExtraSlow` mode. The rationale for this modification was a planned field study for the

purpose of collecting real fault data to be used in generation of mutants. The field study will be described in Section 7.2.7.2.

The complete UML state machine consists of one orthogonal state with two regions. Enclosed in the first region are two simple states and two simple composite states. The simple composite states contain two and three simple states. The second region encloses one simple state and four simple composite states that again consist of, respectively, two, two, two, and three simple states. This adds up to one orthogonal state, 17 simple states, six simple composite states, and a maximum hierarchy level of two. The unflattened state machine contains 61 transitions.

Having both concurrent and hierarchical states, the state machine had to be flattened before being used as input to test case generation. The STATEMACHINEFLATTENER [34] component of TRUST was used for this purpose. In the outset, the flattened model consisted of 82 states and 508 transitions, of which 193 were guarded. However, as addressed above, the flattened model contained infeasible state combinations and transitions as the current version of the STATEMACHINEFLATTENER does not remove these automatically. The user can specify preferences in a provided Kermeta transformation. The outcome of the transformation is a model where these combinations are excluded. After removing infeasible transitions (both due to incompatible state invariants (12 state combination, 145 transitions) and guards that will never become true (14 transitions)), the state machine included 68 simple states (excluding initial and final state) and 349 transitions, of which 107 are guarded. The characteristics of the unflattened and flattened UML state machines are summarized in Table 7. The unflattened state machine can be found in Appendix E.

Table 7 Details of the modified version of the SUT, complete version

State Machine	Composite States	Orthogonal States	Sub Machine States	Simple States	Transitions (guarded)
Hierarchical	6	1 (concurrent regions = 2)	0	17	61 (17)
Flattened	-	-	-	68	349 (107)

Modified Abstract Test Model

To see the effect of having a less precise test model on the cost-effectiveness of the testing strategies, the complete test model was modified. By removing one[3] level in the state hierarchy, the model was abstracted to generate a less complete test model; the contents of every simple composite state were removed.

Raising the abstraction level brought on questions regarding how to set the inclusion criterion for transitions connected to those modified super states. In the end, only transitions that were sourced or targeted in the edge of the super state were kept. This means that the transitions that were targeted in entry points or sourced in exit points contained in the super states were deleted. The explanation for this being that those transitions are not common behaviour to *all* sub states of that super state; such behaviour is only common to those sub states that have incoming transitions to a particular entry point. The same regards outgoing transitions from exit points. Hence, a consequence is that parts of the super-state behaviour are overlooked. Transitions sourced in the edge of a super state, on the other hand, concerns all sub states.

Potentially, to capture more of the SUT behaviour, transitions sourced in exit points belonging to the super state could also be kept. This entails that more of the possible behaviour is tested, though with an increased number of infeasible test cases due to unsatisfied guard conditions. This is particularly true when the guard contains state variables that are specific to the removed sub states. Though not impossible, it is then hard to know upfront whether or not the guard can be satisfied. It would be applicable if analysis of the required path in order to satisfy the guard was implemented.

The outcome of applying the abstraction approach just described was a UML state machine consisting of an initial state, a final state, two transitions, and one orthogonal state with two regions. The first region contained one initial state, five simple states and 14 transitions, whereas the second region contained one initial state, one final state, four simple states and 15 transitions (of which seven were guarded).

Due to the concurrent state, also this model had to be flattened. The flattened version resulted in one initial state, one final state, 25 simple states, and 123 transitions (of which 35 were guarded). After removing four incompatible states and 37 infeasible transitions from the flattened version, the abstract model was left with one initial state, 21 simple states, one final

[3] We can only reduce the model with one level due to the fact that maximum depth in the model is 2.

state and 86 feasible transitions, of which 28 was guarded. The characteristics of the unflattened and flattened UML state machines are summarized in Table 8.

Table 8 Details of the modified version of the SUT, abstract version

State Machine	Composite States	Orthogonal States	Sub-Machine States	Simple States	Transitions (guarded)
Hierarchical	0	1 (concurrent regions = 2)	0	25	123 (35)
Flattened	-	-	-	21	86 (28)

The abstract test model can be found in Appendix F. The resulting code for the SUT consisted of 26 classes and 3372 LOC (1227 h, 2145 cpp) (without blank lines).

Please note that from now on, the modified version of the SUT is referred to as the SUT.

7.2.7.2 (A2) Collecting Fault Data for Creating Mutants

The SBT criteria and oracles were evaluated using mutation analysis. As discussed in Chapter 3, such evaluations have mostly been carried out using artificial faults, i.e. mutation operators. In order to increase the external validity of the results, a field study[4] was conducted for the purpose of collecting real fault data. The collected faults were later used to create mutant versions of the SUT for the purpose of evaluating the SBT strategies. It is important to note that the testing strategies were selected before collecting fault data.

The study was carried out in ABB's departments in Västerås, Baden and Shanghai October/November 2008. Having different UML and domain knowledge, 11 ABB developers were asked to implement a change task to the SUT. They were instructed to modify both the model and code. Participants were strictly asked to work individually. Five participants solved the task in separate rooms, but due to lack of location resources the other six participants had to work in shared offices, two in each office. The author supervised the sessions. The maintenance task itself was initially suggested at a high level by ABB; however, defined and split in six sub tasks by the author of this thesis, and finally approved by the company. The maintenance task consisted of adding an extra gear or mode called `ExtraSlowSpeed` in which industrial machines may be operated. The subjects attended an introductory session where the extended state-design pattern was explained. They were also provided with both

[4] The original goal was to study the effect of using a state-based development method as compared to a more traditional baseline method. To ensure the necessary depth and focus in the thesis within a realistic time horizon, however, the goal of the field study was changed. The main purpose was to gain a better understanding of how efficient state-based testing is in a realistic context with representative change tasks. Therefore, fault data was collected.

textual and graphical documentation, in addition to a manual on how to apply the extended state-design pattern.

A correct version of the model and code, representing the SUT after implementing the maintenance task, was made by the author and exhaustively tested with each of the coverage criteria in focus of this study. The faults extracted from the field experiment data, immediately introduced, were inserted into this correctly modified version of the SUT.

The 11[5] solutions were manually inspected by the author. In total, 26 faults were detected from code inspections. Note that because the objective was to compare the fault-detection effectiveness of the testing criteria, it is crucial that the seeded faults do not cause compilation errors. This means that only *logical faults* that could not be detected by the compiler were selected. The extracted fault data were used to create 26 faulty versions of the code by seeding one fault per program. Among these faults, 11 were sneak paths [31] (i.e., implicit illegal transitions present in the state machine). To detect faults of this type, the model should reflect the preferred behavior of the system when exposed to unexpected events. At this stage, our model had no support for the handling of such unexpected behavior. Hence, as sneak paths could not be caught by any conformance test suite generated from the model, 15 out of 26 mutant programs were detectable by the conformance test suites.

The faults from the code reflect the following modeling errors:

- *Missing transitions:* An expected guarded transition triggered by a completion event from `ExtraSlowConfirm` to `ExtraSlowConfirmed` was missing from the model. One of the subjects only accounted for the transition that was explicitly triggered by the `ClearEnableDevice()` event. The subject did not consider the transition that would fire if `enableDevice` already was false.
- *Additional transitions (sneak paths):* Another subject erroneously added transitions from `ExtraSlow` not only, as specified, to `Manual`, but also to `Auto` and `ManualFullSpeed` and vice versa.
- *Guards that were not correctly updated:* One of the subjects added a clause in the guard on the transition from `Disabled` state to `Active` state so that a transition would be fired only if the mode is `ExtraSlow` or the speed is `extraSlow`.
- *Guards that were modified which should not have been changed:* In one of the erroneous versions, the event handling operation `SetModeSoftStop()`*'s* guard

[5] The field experiment originally had 11 participants. However, as one of the participants made no modifications to the code, that particular solution was considered as irrelevant for this study.

was modified so that the state machine would transition from state `Active` to state `SoftStop` only if the mode is `ExtraSlow`. In a correct version, however, the state machine should transition regardless of the mode.

- *Erroneous on-entry behavior*: An operation that handles the on-entry behaviour of the super state `ExtraSlow` was introduced with an error: the sub states were updated with the same value for state variables. The only common value, however, should be given the `speed` variable; `blockDriveEnable` should be true in the `ExtraSlowConfirm` state and false in the `ExtraSlowConfirmed` state.
- *Incorrect state invariants:* The state variable `blockDriveEnable` was missing from `ExtraSlow`*'s* state invariant.
- *Missing on-entry behavior:* `OnEntry()` was not added for `ExtraSlow`.

Figure 13 shows one of the mutants where updating a variable (`blockDriveEnable`) as part of the `onEntry` action was left out.

```
void ExtraSlowConfirm::OnEntry(SafetyBoardControl *myControl)
{
    ExtraSlow::OnEntry(myControl);

    //fault 12 missing code: myControl->WriteBlockDriveEnable(true);
    myControl->WriteSpeed(SlowSlow);

    if(!myControl->GetEnableDevice())
    {
        ChangeState(myControl, ExtraSlowConfirmedState);
    }
}
```

Figure 13 Code showing a mutant version of the SUT

7.2.7.3 (A3) Extending and Configuring the Tool

In order to use the model-based tool TRUST in this study, it had to be extended and configured to meet the requirements and conditions of the investigation. TRUST was extended to support the relevant coverage criteria and to support two oracles. Moreover, the concrete test-case generator was extended for producing C++ code, in addition to providing a test environment that facilitates selection of values for externally controlled variables in guard conditions. Finally, TRUST was extended with support for sneak-path testing.

Extending TRUST with Coverage Criteria

The first version of TRUST supported generation of test cases according to the TT criterion [22], which is a modified, stronger version of RTP, and for AT. The TT criterion was implemented following the breadth-first algorithm. However, only one test case was generated for each transition, and hence only useful for models with guard conditions without OR clauses. The AT criterion was implemented using a depth first traversal of the state machine.

Unfortunately, generation of TT test suites from the SUT using the first version of TRUST led to Eclipse running out of memory. It was thus decided to use the weaker stopping criterion for RTP as first proposed by Binder [31]. Two versions of RTP were implemented following the breadth-first algorithm: (1) a weak version where guard conditions are traversed once, and (2) a stronger version where a guarded transition is traversed as many times as there are OR clauses in the condition. Also the ATP was implemented using a breadth-first algorithm. For LN2, LN3, and LN4, transition trees were generated by traversing respectively two, three, and four levels from initial state using breadth-first traversal of the state machine.

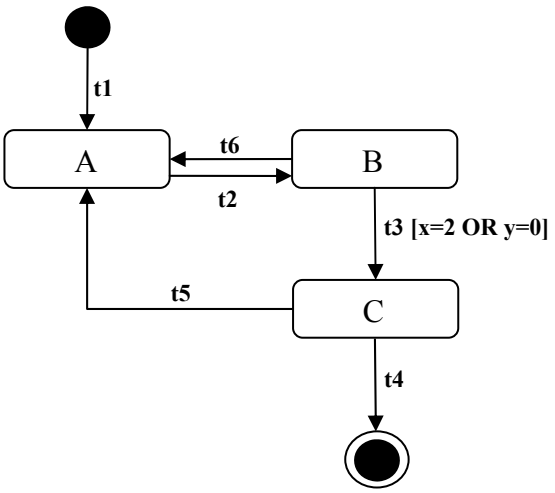


Figure 14 Example state machine

To illustrate the scope of the criteria, the following sequences of transitions are generated from the example model shown in Figure 14:

- AT: {(t1, t2, t6); (t1, t2, t3, t4); (t1, t2, t3, t5)}
- RTP: {(t1, t2, t6); (t1, t2, t3[x=2], t4); (t1, t2, t3[x=2], t5); (t1, t2, t3[y=0])}

- RTP weak[6]: {(t1, t2, t6); (t1, t2, t3, t4); (t1, t2, t3, t5)}
- ATP: {(t1, t2, t6, t2); (t1, t2, t3, t4); (t1, t2, t3, t5)}
- LN2: {(t1,t2, t6); (t1, t2, t3)}
- LN3: {(t1, t2, t3, t4); (t1, t2, t6, t2); (t1, t2, t3, t5)}
- LN4:{(t1, t2, t3, t4); (t1, t2, t3, t5, t2); (t1, t2, t6, t2, t6); (t1, t2, t6, t2, t3)}

Extending TRUST with another Oracle

Differing in the level of details that are checked, two oracles were evaluated: oracle O1 checks that the pointer to the expected state is correct, in addition to evaluating the state invariant; whereas oracle O2 only checks that the pointer to the state after some event has occurred is as expected. This means that O1 is stronger than O2; an expected result would be that O1 is more expensive than O2. What is to be shown is the effect such a reduction in oracle strength has on the cost and effectiveness.

Extending TRUST with Support for Concrete C++ Test Cases

Since the first instantiation of TRUST was implemented for another case supporting a Python-based scripting language, TRUST needed to be extended to support SBT for the SUT in focus of this thesis. The SUT was implemented in C++ and thus a language in favor when testing the SUT in terms of efficient interaction between the SUT and the generated test cases. TRUST was extended to interact with a new *TestscriptGenerator* component, i.e., a new test script generator supporting C++. Extending TRUST for C++ involved adding a new set of transformation rules according to C++ syntax to the *TestscriptGenerator* component of TRUST. This required changes in the MOFScript rules. There was some reuse of the MOFScript rules used in the tool instantiation for the first version of TRUST. Only the logic for traversing the tree could be reused, however, because the mapping rules for C++ were quite different from the Python-based scripting language already supported due to different language constructs. Test data was in the outset generated by randomly selecting the value for the parameters based on the data type. Java Native Interface (JNI) [118] was used to access EOS from the C++ test scripts.

Support for Selecting Appropriate Environment Values

This section addresses infeasible transitions and how these were handled. There are several reasons for a transition being considered as infeasible. Firstly, in concurrent systems, certain

[6] The weak version of RTP gives the same coverage as AT and are thus not further investigated.

state combinations may be impossible due to conflicting state invariants. The belonging incoming and outgoing transitions sourced and targeted in such combinations, will also be infeasible as those states will never be reached. The incorrect state combinations must be removed together with the incoming and outgoing transitions as tests including these transitions expectantly will never pass. Secondly, the value combination of test data in guards is a common reason for why test cases fail. Such test cases can also be considered as infeasible because as long as they remain unhandled, these test cases are in fact infeasible. Obviously, the 'wrong' combination causes test cases to fail – even though the system reacts as intended. The infeasible test cases may thus have a negative impact on the percentage of the achieved coverage criterion as certain transitions will not be fired. According to Binder, 10–15 percent is a typical allowance for reduced coverage amongst others due to infeasible test paths [31].

For testers highly familiar to the SUT, the first type of infeasible transitions is easy to address. The second, however, requires handling of test data to manipulate external variables that controls the firing of transitions.

The initial instantiation of TRUST supported static test-data generation. However, as approximately one third of the transitions in the SUT were guarded, test-case execution showed that hardly any test cases passed. Poor results were obtained in spite of having infeasible state combinations and the belonging transitions removed. The results thus point in the direction of presence of infeasible transitions due to unsatisfied guard conditions. The reason for this is that the provided test data did not satisfy the many guards that relied on *externally controlled variables*. As a consequence, test cases expecting the system to transition from one state to another, failed. Hence, the results were not of great interest as it was impossible, without analysis, to identify whether a test failed due to a real bug or due to unsatisfied guard conditions. Manual inspections of all test cases were not applicable due to the large sizes of the test suites (from 27–1,425 test cases). To illustrate this issue, for the RTP test suite, as much as 88 percent of the test cases failed due to unsatisfied guard conditions. Therefore, to reduce this type of infeasibility and to serve interesting test results, TRUST was extended to support intelligent test-data generation – more precisely to provide test data that satisfy guard conditions. Automated test-data generation has shown good results for identifying dynamic test data (e.g., [125]). In this study, however, the dynamic test-data generation was hard coded due to limited time resources.

As discussed in this section, model-based testing may not be able to achieve 100 percent coverage of the criterion we request due to infeasible transitions. However, by providing

appropriate test data (when possible), the number of infeasible test cases will be reduced and thus increase the quality of the test suites.

Extending TRUST to Support Sneak-Path Testing

In order to catch sneak paths, it is necessary to augment the test suites with sneak-path testing. For each state in the SUT, all possible events not specified for the particular state are invoked. The correct behaviour would be for the SUT to remain in its current state. This technique will catch faults that introduce undesired additional behaviour, in terms of extra transitions, other than what is specified in the UML state machine.

The sneak-path test suite was generated using Kermeta. One UML state machine was created for each state and its expected behaviour to unexpected events. To be able to position the SUT in the state to be tested, each state machine also contains the states and transitions from initial state that makes a path to the state under test. The AT coverage criterion was used to generate the test trees. MOFScript was, as for the previously generated test suites, used to create the concrete test cases. Sneak-path testing was applied to the abstract and complete model, using both oracles.

7.2.7.4 (A4) Generating Test Suites

The extended version of TRUST was then configured and used to automatically generate executable test suites from the two test models previously introduced. The prepared test models were used as input models to TRUST. As the state-based criteria are defined for finite state machines, a prerequisite for generating the test suites is to use an input state machine without concurrency and hierarchy. The first step in TRUST was thus to *flatten* [31] the test model, i.e., removing hierarchy and concurrency from the model as previously described.

To create the *abstract test cases*, in the shape of a test tree, each of the algorithms for obtaining test suites satisfying the coverage criteria under study were applied on the flattened state machine. TRUST then created *concrete test cases* using MOFScript which took the flattened state machine in addition to the generated test tree as input. In MOFScript, the test tree was traversed path by path – each path produced one test case. A separate C++ file was created for each test case. A main C++ file was generated where each of the test cases were invoked. Each test case was invoked with a new object of the SUT.

The test suites were executed on what is considered being a correct version of the code, i.e., one which does not cause the test suites to detect failures. Results were then analyzed in order to remove actual infeasible test cases and to resolve infeasible transitions caused by

unsatisfied guard conditions due to externally controlled variables. The latter issue was handled by providing an environment which enables transitions to be fired (see previous section), and then re-generating the concrete test cases. When the test suites executed successfully, the test suites were run on the mutant versions of the SUT.

The following steps summarize the automated experimental process that was followed:

Step 1. Flatten[7] input state machine.

Step 2. Select test adequacy criterion.

Step 3. Construct abstract test cases in the form of a test tree. The algorithms used to traverse the state machine are previously described.

Step 4. Select oracle.

Step 5. Traverse the tree and select test data to generate concrete test cases. One test suite is generated per tree.

Step 6. Build and execute the test suite on the correct version. Ensure that test results reveal no errors. Then build and execute the test suite on each of the 26 mutant programs.

Step 7. Analyze test results provided by the state-invariant oracle on all mutants.

7.2.7.5 (A5) Executing Test Suites

Handling Weaknesses in the Implementation

During preparations of the sneak-path testing, some weaknesses were detected in the implementation of the SUT. These weaknesses had to be resolved in order to run the sneak-path test suite correctly.

Recall that the physical mode switch was represented in the SUT by an array called the `mode enum`. According to the specification of the SUT, it should only be possible to go from mode `manual` to mode `extra slow speed` and vice versa. For instance, it should not be allowed to go directly from `manual full speed` to `extra slow speed`. Eleven of the seeded faults reflect such mistakes. When running the sneak-path test suite on a correct version of the SUT (without sneak paths implemented in the code), however, it was discovered that the test suite erroneously caused changes in the `mode enum`. That is, as a response to the sneak-path test suites's attempt of invoking unexpected behavior, the implementation made changes to the `mode enum` even though the physical switch was not actually carried out. This means that the implementation would set the mode switch regardless

[7] The test model only needs to be flattened once.

of the transition being fired or not. The change in mode switch status should take place *inside* the event handling operation – not before.

This also regards the change in the value of the variable `blockDriveEnable`; its value was set regardless of the transition being fired or not. Updating the value of this particular variable should have taken place in the event handling operation – not before it was invoked.

The third weakness that was detected regards the event `reset()`. In the original code, it was implemented as part of the `DriveEnable` super state – implying that all sub states have this event handling operation available. It should, however, be specific to the `SoftHalt` state and thus only implemented in the concrete `SoftHalt` class.

A similar weakness was found in relation to the event `StartInAuto()`. Again, this should have been implemented as a sub state specific event handling operation for the concrete class `IdentifyingMode`.

Practical Issues

To automate the build, execution, and timing when executing the test suites on the correct and mutated programs, batch files were created for each test criterion, for each oracle. The version of the SUT to be executed was copied into the Visual Studio project folder. The project was then rebuilt and executed. One result file in the format of a text file was created for each version of each coverage criterion. The result file contains the results for the correct version and the 15 mutants.

Replicating the Experiment

For certain coverage criteria, like AT, RTP, and ATP, the generation of test trees from state machines is not deterministic as several test trees can satisfy the criterion. The explanation for this is that the structure of the tree depends on the sequence of the selected outgoing transitions when traversing the state machine. However, due to possible differences in the fault-detection level of the various test suites, the results of executing different test suites that fulfill the same test criterion may differ. Such random variations in the results were accounted for by repeating the experiment 30 times; 30 different trees were created using random selection of the order of outgoing transitions from states to generate distinct test suites. Being selected without replacement from the population of all possible trees that achieves each of the criteria, only trees distinct from the selected trees were added to the selection. Briand *et al.* [81] explain the variation in fault-detection ability in the following way: “In fact, even though transition trees execute the same methods, those methods are executed in different orders, thus

satisfying preconditions in different ways and leading to different outputs and state changes as specified in different conjuncts of the post conditions."

The test suites were obtained by traversing each of the 30 test trees covering all paths, and executed on the mutant programs.

7.2.8 Analysis Procedures

Please recall that the main objective of this research (Section 7.2.5) was to evaluate the cost-effectiveness of SBT when varying coverage criteria, oracle, and test model abstraction level. Data was collected on the surrogate measures for cost and effectiveness provided in Table 9, further described in Section 7.2.7. Hence, the main focus of the analysis was to identify trends and significant results on cost-effectiveness of SBT.

Table 9 Features of variables

Surrogate Measures for Cost	
Dependent Variable	**Accuracy of Measurement**
Test generation time	Continuous
Test execution time	Continuous
Test-suite size	Discrete
Surrogate Measures for Effect	
Dependent Variable	**Accuracy of Measurement**
Mutation score	Continuous

The analysis procedures are based on both quantitative and qualitative data; quantitative data were used to explore and compare different aspects of state-based testing, whereas qualitative data were used to explain further obtained results from analyses of the quantitative data. The tools JMP 7 [126], Excel 2007, and R [127] were used for data analysis.

7.2.8.1 Quantitative Analyses

Statistical significance [128] provides the probability that differences between observations actually exist. This means that the groups being compared differ to a greater degree than would be expected by chance. The conventional level of 5 percent ($\alpha = 0.05$) was chosen as significance level in this thesis. Moreover, to describe the magnitude and direction of the observed differences, the effect size [129] was also provided.

Descriptive Statistics

The main features of the collected data, like central tendency, statistical variability, and distribution shape, were described using descriptive statistics. The summary statistics (the *five-number summary)* and the associated box plot were used for this purpose.

Cost-Effectiveness

The cost-effectiveness of SBT, represented by the dependent variables, was compared among the three independent variables, visualized by graphs showing how the cost and effectiveness change when varying the independent variables. The graphs enable comparisons of the strength of killing mutants among testing strategies at which cost. For each dependent variable, there are 360 data points (15 mutants, 6 coverage criteria, 2 oracles, and 2 test models).

Potential random variations in the mutation score across test suites for a given coverage criterion creates a need for analysing the distribution of the mutation scores. Furthermore, as the collected data was found to be non-normally distributed, the Wilcoxon signed-rank test [128] was used to provide non-parametric statistical hypothesis tests of whether or not the dependent variables (preparation time, execution time, and mutation score) are significantly different. Tests were run on collected data from generating and executing 30 test suites of each criterion.

By this analysis, we can present an overview of the expected percentage of mutants that are killed by the testing strategies (including coverage criteria and oracles) at different test model abstraction levels that implies different cost.

7.2.8.2 Qualitative Analyses

Expectantly, certain mutants could not be killed by every combinations of coverage criteria, oracle and test model. Analyses of un-killed mutants were performed to identify mutants that were difficult to kill.

7.2.9 Validity Procedures

The four validity categories as suggested by Runeson and Höst [122] to be used in case studies are discussed in this section.

Construct validity regards to what extent the operational measure that are studied really represent what the researcher have in mind and what is investigated according to the research questions [122]. That is, it concerns the establishment of correct operational measures for the concepts being studied. This validity issue was handled using multiple sources during data

collection. Several surrogate measures were selected to describe cost-effectiveness of SBT. Also, most of these measures are commonly known measures in cost-effectiveness studies.

Internal validity is of concern when causal relations are examined [122]. Investigated variables may also be affected by extraneous variables, confounding factors, not accounted for in the study, thus a threat to the internal validity. What is observed should be attributed to the studied variable and not to potential confounding factors. One detected risk in terms of internal validity was the possible randomness in the obtained results for three of the coverage criteria. This issue was handled by generating 30 test trees for those coverage criteria, thus replicating the experiment for these criteria 30 times. Statistical hypothesis testing was applied to the collected data.

External validity is concerned with to what extent it is possible to generalize the findings, and to what extent the findings are of interest to other people outside the investigated case [122], i.e., to whether a domain exists to which the results are relevant. How is it possible to generalize from a single case? Yin [64, p. 10] provides the following answer:

> *The short answer is that case studies, like experiments, are generalizable to theoretical propositions and not to populations or universes. In this sense, the case study, like the experiment, does not represent a "sample", and in doing a case study, your goal will be to expand and generalize theories (analytic generalization) and not to enumerate frequencies (statistical generalization).*

As stated by Runeson and Höst [122], for case studies, the intention is to enable analytical generalization where the results are extended to cases which have common characteristics and hence for which the findings are relevant, i.e., defining a theory.

The main strength of this study is, in fact, its external validity. The system in focus of this thesis is highly representative of systems with state-based behavior thus improving the external validity. It is important to provide detailed context descriptions, like system characteristic, development and testing procedures, such that others can relate the results to their own context. This information is provided in sections 7.2.6 and 7.2.7. Moreover, in contrast to the majority of existing studies using artificial faults in such evaluations, the faults used in the evaluation of SBT are real faults collected in a field study conducted in ABB (Section 7.2.7.2). The study design of the thesis is based on existing theory, and the results from these studies are compared to the results of this thesis.

A threat to the external validity could be that one researcher was the subject in the case studies. The rationale for this being lack of resources and a general lack of state-based testing experience in the company. However, as the test case generation process was automated as compared to other studies where test cases are manually generated, this is not considered being a threat.

Reliability concerns to what extent the data and the analysis are dependent on the specific researchers [122], e.g., unclear descriptions of data collection procedures such that later replications of the study could give different results. This is addressed by providing as detailed design and analysis procedures as possible.

Next, the four case studies in Chapter 8–11 present cost-effectiveness analyses for the purpose of answering the research questions stated in Section 7.1.2.1 based on the data collected according to procedures described in Section 7.1.2.3.

8 Case Study 1 – What is the Cost-Effectiveness of the State-Based Coverage Criteria All Transitions, All Round-Trip Paths, All Transition Pairs, Paths of Length 2, Paths of Length 3, and Paths of Length 4?

Based on the case-study design in Chapter 7, this chapter presents results from a case study which addresses RQ1 regarding the cost-effectiveness of the six state-based coverage criteria:

> *RQ1: What is the cost-effectiveness of the state-based coverage criteria all transitions (AT), all round-trip paths (RTP), all transition pairs (ATP), paths of length 2 (LN2), paths of length 3 (LN3) and paths of length 4 (LN4)?*

A number of figures are provided, visualizing how the dependent variables mutation score, test-suite size, and test suite preparation and execution time are affected by the different coverage criteria. Section 8.1 provides descriptive statistics for the collected data on cost and effectiveness, whereas Section 8.2 regards statistical tests applied on the data from Section 8.1. A cost-effectiveness analysis is presented in Section 8.3. A qualitative analysis of undetected mutants is provided in Section 8.4. Results are discussed towards existing research in Section 8.5. Furthermore, Section 8.6 discusses results from related work against results from this study. Finally, Section 8.7 summarizes this chapter.

8.1 Descriptive Statistics

8.1.5 Descriptive Statistics for Cost

This section presents main features of the collected data on cost: the test-suite sizes and the time spent on preparing and executing these test suites.

8.1.5.1 Test-Suite Size

First of all, consider Table 10 which lists the test-suite sizes for the six testing strategies. For now, the coverage criteria were applied on the complete test model using the strongest oracle, O1. The reader is referred to Chapter 7 for details on the complete test model and oracle O1. As we can see, LN2 resulted in 27 test cases – the smallest test suite among the studied coverage criteria. More than five times larger, LN3 contained 143 test cases, closely followed by AT with 166 test cases. Approximately 50 percent increase was shown for RTP with 299 test cases. Again, a major increase in size: LN4 provided 764 test cases. Finally, ATP resulted in the largest test suite with 1,425 test cases.

Table 10 Test-suite sizes – ascending order[8]

COVERAGE CRITERION	TEST MODEL	ORACLE	TEST-SUITE SIZE
LN2	**complete**	**O1**	**27**
LN3	**complete**	**O1**	**143**
AT	**complete**	**O1**	**166**
RTP	**complete**	**O1**	**299**
LN4	**complete**	**O1**	**764**
ATP	**complete**	**O1**	**1,425**

It was expected that LN2, LN3, and AT provided small test suites – having their definitions in mind. More important to notice, however, is the large difference between the smallest and the largest test suite: ATP is almost 53 times larger than LN2 – which must be considered being a significant increase in cost.

Figure 15 shows the distribution of test-suite sizes. The corresponding box plot is shown in Figure 16.

[8] Please recall the following regarding AT, RTP, and ATP: Numbers are the average of 30 replications.

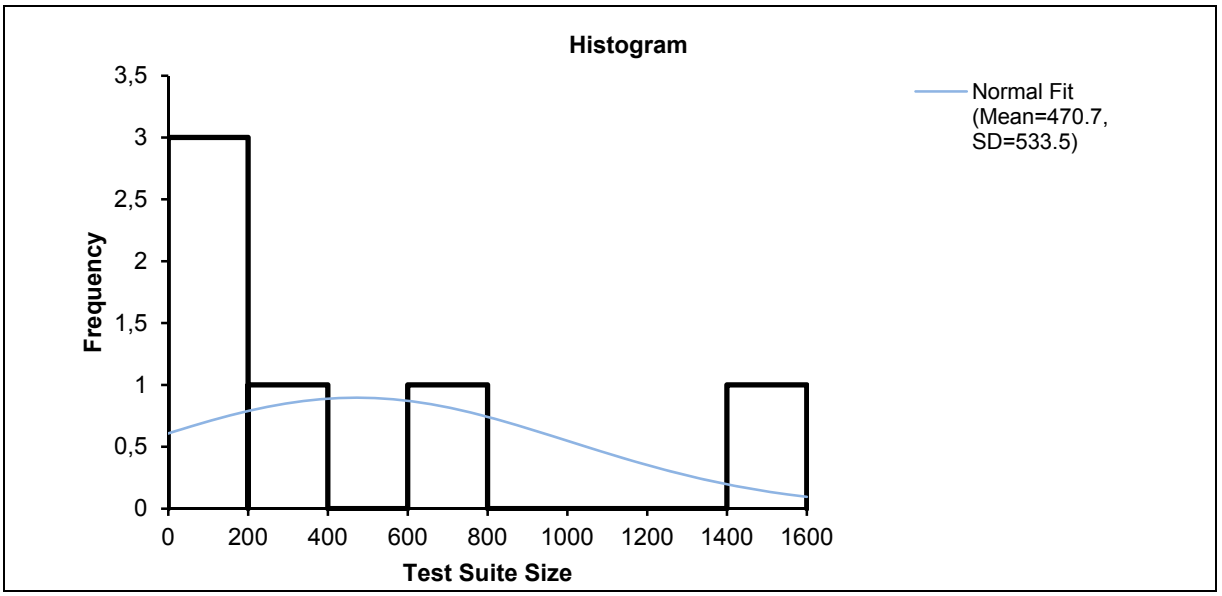


Figure 15 Distribution – test-suite size

Also worth noticing is the difference in mean (470.7 test cases) versus median (232.4 test cases). This is explained by the distribution being skewed to the right by ATP.

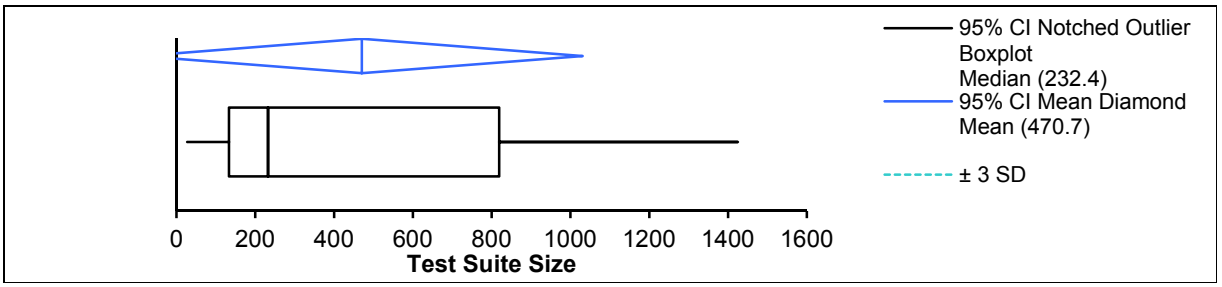


Figure 16 Tendency, dispersion and distribution shape of the test-suite sizes

From the box plot, it is even clearer that the majority of the testing strategies consist of less than 800 test cases. Only one testing strategy had a higher number of test cases – as we can see from Table 10, this regards ATP with 1,425 test cases.

8.1.5.2 Time

Let us look at the second surrogate measure of cost – time, split in *preparation* and *execution* time. Descriptive statistics are displayed in Table 11 and Table 12. The statistics include the minimum, the 25 percent quartile, the mean, the median, the 75 percent quartile, the maximum, and the standard deviation in the collected data for each coverage criterion. Timing data was collected by running the experiment on a Windows 7 machine with an Intel(R)

Core(TM)2 Duo CPU P9400 @ 2.4 GHz processor, and with 2.4 GB memory. Note that time is measured in seconds.

Looking at the mean values, we observe that LN2 had the lowest values for preparation time (126 seconds). LN3 had the second lowest value, 509 seconds. RTP ranked third with 531 seconds. An enormous increase, almost 87 percent, was seen between preparation time for RTP and AT (3,995 seconds). The second highest measure collected for preparation time was LN4 with 5,295 seconds. The highest value, 28,819 seconds, was observed for ATP.

Table 11 Descriptive statistics - preparation time

Strategy	Model	Oracle	Min	Q1	Mean	Median	Q3	Max	St Dev	N
LN2	Complete	O1	126	126	126	126	126	126	-	1
LN3	Complete	O1	509	509	509	509	509	509	-	1
RTP	Complete	O1	484	512	531	525	545	607	28	30
AT	Complete	O1	2,506	2,763	3,995	3,013	3,665	15,939	3,264	30
LN4	Complete	O1	5,295	5,295	5,295	5,295	5,295	5,295	-	1
ATP	Complete	O1	28,377	28,576	28,819	28,797	29,028	29,398	273	30

Results show for execution time that, again, LN2 provided the lowest value (18 seconds). LN3 was measured to 136 seconds, followed by RTP with 489 seconds. Almost doubling (more precisely, 1.74 times larger) the time seen for RTP, LN4 was measured to use 850 seconds on executing the test suite. The second highest time was measured for AT – 2,455 seconds. Finally, execution of the ATP test suite took 3,341 seconds.

Table 12 Descriptive statistics - execution time

Strategy	Model	Oracle	Min	Q1	Mean	Median	Q3	Max	St Dev	N
LN2	Complete	O1	18	18	18	18	18	18	-	1
LN3	Complete	O1	136	136	136	136	136	136	-	1
RTP	Complete	O1	341	460	489	504	524	607	54	30
LN4	Complete	O1	850	850	850	850	850	850	-	1
AT	Complete	O1	1,765	2,108	2,455	2,377	2,785	3,617	469	30
ATP	Complete	O1	2,607	2,972	3,341	3,381	3,669	3,978	429	30

Interestingly, the correlation between preparation and execution time was not consistent. As we can see from the results, the mean of LN4 was higher than the measured time for AT, what comes to preparation time. Regarding execution time, however, AT had a higher measured value than LN4. This can be explained by the way the state machine was traversed when generating the AT and LN4 test suites. AT was generated using a depth first algorithm, which typically results in few, but long test cases. LN4, on the other hand, was generated by breadth first, providing many, but short test cases. The time spent on generating the many LN4 test cases would thus take longer time than generating fewer, but longer AT test cases. Imagine the additional code required in separate test cases as compared to having one long test case. Although LN4 is approximately 4.6 times larger than AT in terms of number of test cases, the execution time takes five times longer (2,455 seconds (LN4) versus 489 seconds (RTP)) due to the nature of the AT test cases.

Figure 17 displays a graphical representation of the means. The figure shows a significant increase in preparation time for ATP.

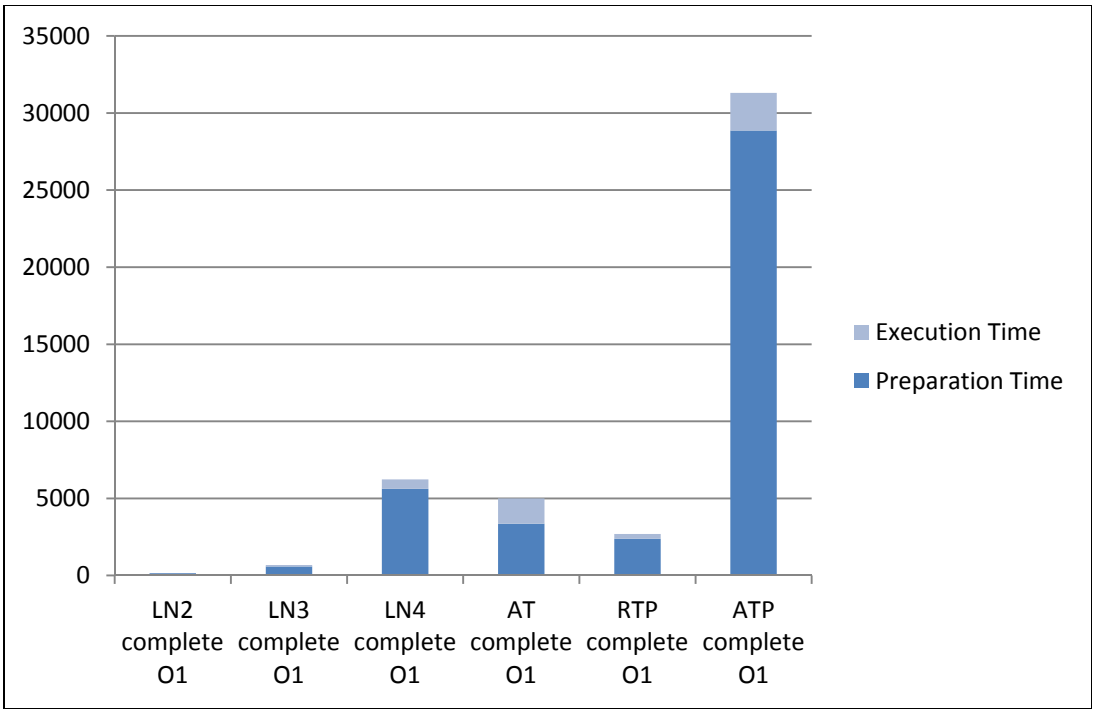


Figure 17 Preparation and execution time in seconds shown for each coverage criterion

The figures displayed in Table 11 and Table 12 illustrate the variations in results among the generated trees within each of the strategies AT, RTP, and ATP. This will be further addressed in the following paragraphs.

Comparing Means

As described in the design chapter (Chapter 7), the collected data presented in the previous sections are, for each of AT, RTP, and ATP, based on 30 replications of the experiment. Recall that the trees from which the test suites are generated were randomly selected without replacement to the population of trees within each strategy. The mean is expected to be an estimate of the real value μ of the population, and closer to μ than each of the trees independently. Therefore, to reduce the uncertainty in the mean, a number of 30 trees were selected for AT, RTP, and ATP. This section investigates those replications for the purpose of identifying possible statistical variability.

The observations for each group (i.e., combinations of coverage criterion, test model and oracle) are independent. Being selected without replacement from the population of all possible trees that achieves each of the criteria, only trees distinct from the selected trees were added to the selection.

The histograms and box plots in Figure 18 and Figure 19 show the distribution of the time spent to prepare and execute the test suites, respectively. The data material can be found in Appendix G.

Figure 18 presents the performance of AT, RTP, and ATP regarding preparation time. None of the histograms show a normal distribution.

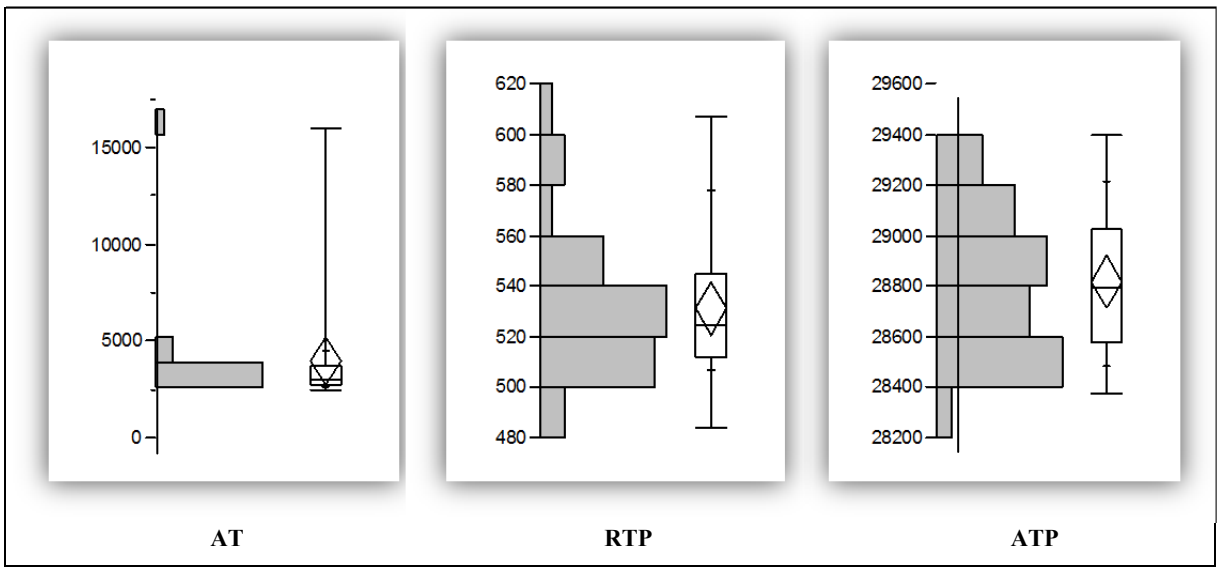


Figure 18 Distribution – preparation time

Starting with AT, we see a distribution skewed to the left having the minimum value 2,506 seconds, the mean at 3,995 seconds, and the maximum value at 15,939 seconds. Worth noticing is the relatively significant difference in mean and median, which can be explained

by the fact that there were two outliers in the collected data. Looking at the inter quartile range (IQR) and the belonging upper and lower limits (see Table 13) for what is considered being outliers, we see that 15,781 seconds and 15,939 seconds are outliers as their values exceed the upper limit (5,017.8).

Next, the distribution shown for RTP is skewed to the left. The minimum value was 484, the mean was 531 seconds, and the maximum observed value was 607 seconds. A quick glance at Table 13 shows that, also for this strategy, there was an outlier present – observations above 595 seconds should be considered as outliers, i.e., the observation 607 seconds. Outliers in the lowest quartile would have to be lower than 462 seconds. Such outliers were not present.

Finally, ATP had 28,377 seconds as the minimum value, 28,819 seconds as the mean, and the highest measured value was 29,398 seconds. According to the IQR, outliers would be present if observations were seen above 29,706 or below 27,898. However, all observations were within the IQR from Q1 and Q3, and thus no outliers.

Table 13 Inter quartile ranges – preparation time

Strategy	Q1	Q3	Q3-Q1	IQR	Lower limit	Upper limit
AT	2,763	3,665	902	1,353	1,410	5,018
RTP	512	545	33	50	462	595
ATP	28,576	29,028	452	678	27,898	29,706

Nor Figure 19, which displays the spread in the collected execution-time data, presents normally distributed observations. The execution time for AT varies from 1,765 seconds to 3,617 seconds, for RTP we see a range from 341 seconds to 607 seconds, and for ATP the time is spread from 2,607 seconds to 3,978 seconds.

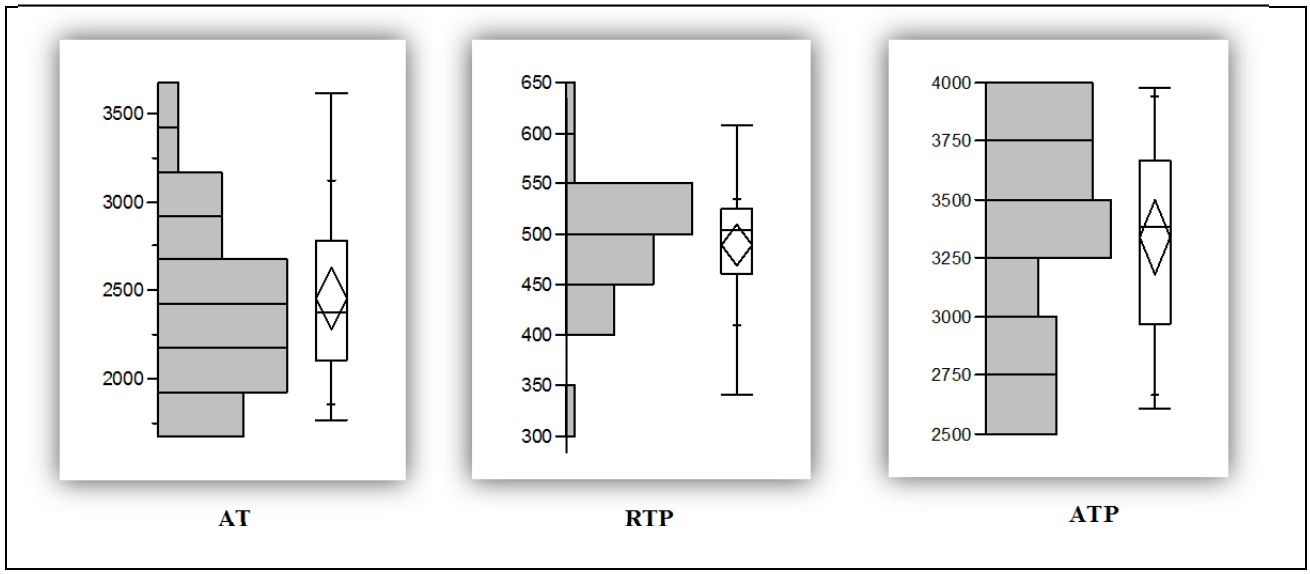


Figure 19 Distribution – execution time

AT's distribution is skewed to the left. The minimum value among the observations was 1,765 seconds, the mean was 2,455 seconds, and the maximum value was 3,617 seconds. There are no outliers observed.

The RTP distribution is slightly skewed to the left. Looking at Table 12, we see that the minimum value was 341 seconds, the mean was 489 seconds, and the maximum value was 607 seconds. Being the minimum value (341 seconds), this observation is an outlier as it exceeds the lower limit (see Table 14).

Finally, the collected data for ATP was rather equally distributed. The mean was at 3,341 seconds, the minimum at 2,607 seconds, and the maximum at 3,978 seconds. No outliers were observed for this strategy.

Table 14 Inter quartile ranges – execution time

Strategy	Q1	Q3	Q3-Q1	IQR	Lower limit	Upper limit
AT	2,108	2,785	676	1,014	1,094	3,799
RTP	460	524	64	96	364	621
ATP	2,972	3,669	698	1,046	1,925	4,715

8.1.6 Descriptive Statistics for Effectiveness

This section describes main features of the collected data on effect, which regards mutation score – the surrogate measure of effectiveness. Table 16 presents the five-number summary. Again, the summary includes the minimum, the 25 percent quartile, the median, the 75 percent quartile, the maximum, the mean, and the standard deviation in the collected data for each combination of coverage criterion, test model, and oracle.

The minimum mutation score (0.333) was observed for LN2. The maximum mutation score (1), however, was reached by AT, RTP, ATP, and LN4. Considering the mutation score results ranked by mean from low to high, the summary shows that LN2 performs significantly poorer than the other coverage criteria. A large gap was to be found between LN2 and the next result – LN3 with 0.933. Quite similar, AT resulted in a high mutation score, 0.997. The best mean value observed for mutation score came with RTP, ATP, and LN4. Explained by similar results within each coverage criteria, the medians closely follow the means. This particularly regards LN2, LN3, and LN4 that were only executed with one test suite. Recall that AT, RTP, and ATP were replicated 30 times due to possible variations in the results regarding several traversal paths providing different trees. This also explains the low standard deviation figures, at least for LN2, LN3, and LN4. Moreover, AT, RTP, and ATP provided high mutation score means across all 30 replications, and hence, low standard deviations.

Table 15 Descriptive statistics for mutation score

Strategy	Model	Oracle	Min	Q1	Mean	Median	Q3	Max	Std Dev	Std Error Mean	N
AT	complete	O1	0.900	1.000	0.997	1.000	1.000	1.000	0.018	0.003	30
RTP	complete	O1	1.000	1.000	1.000	1.000	1.000	1.000	0.000	0.000	30
ATP	complete	O1	1.000	1.000	1.000	1.000	1.000	1.000	0.000	0.000	30
LN2	complete	O1	0.333	0.333	0.333	0.333	0.333	0.333	0.000	0.000	1
LN3	complete	O1	0.933	0.933	0.933	0.933	0.933	0.933	0.000	0.000	1
LN4	complete	O1	1.000	1.000	1.000	1.000	1.000	1.000	0.000	0.000	1

The mutation-score means displayed in Table 15 are graphically visualized in Figure 20. We see that the complete test model combined with oracle O1 resulted in an overall high mutation score, except from LN2.

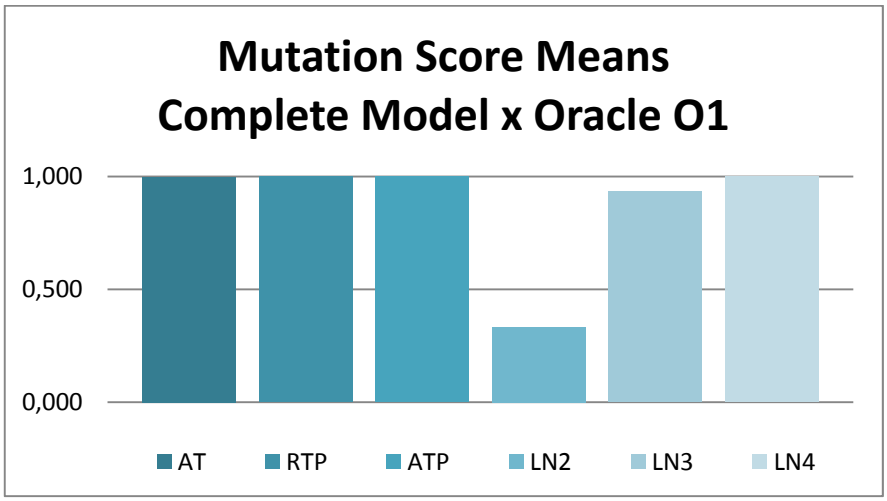


Figure 20 Mutation score means

Comparing Means

This section investigates the replications of AT, RTP, and ATP for the purpose of identifying possible statistical variability in the collected data on mutation scores. Studying the distribution of the mutation score in Table 15 and Figure 21, we see hardly any variation in the observations. The histograms show consistent distributions of the mutation scores. The data material can be found in Appendix G.

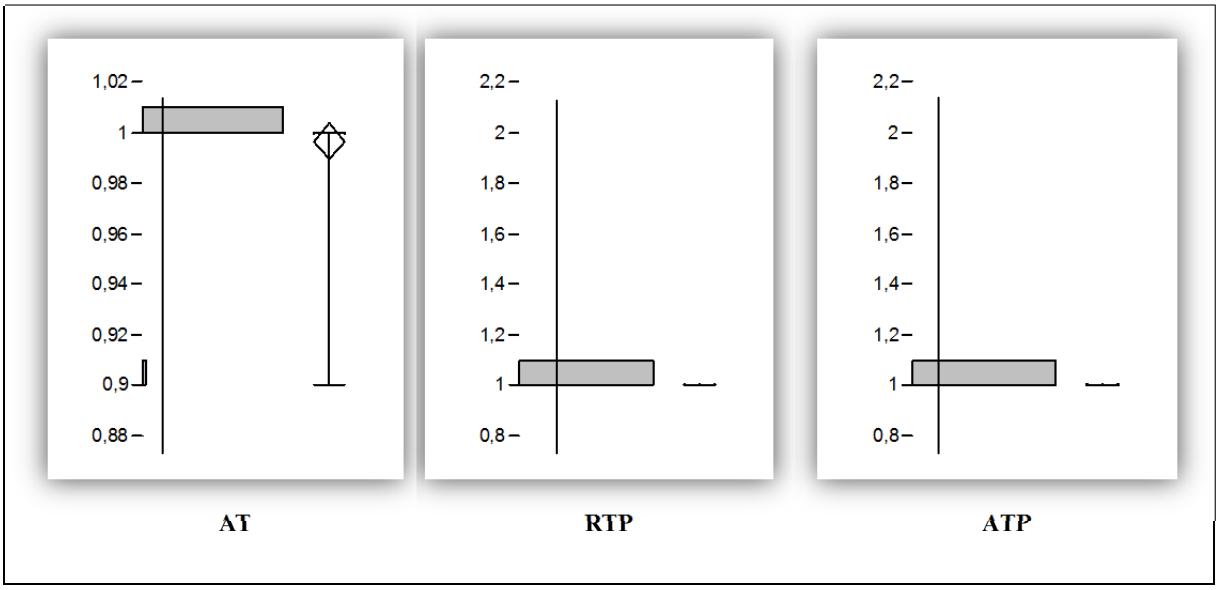


Figure 21 Distribution – mutation score

From the box plots of RTP and ATP, we see that 100 percent of the test suites achieved mutation score means equal to 1. For AT, 29 of the 30 test suites killed every mutant. The remaining test suite killed 14/15 seeded errors.

The next section will present results from tests on significant statistical differences in the replicated data.

8.2 Statistical Tests

This section provides statistical tests on the data described by descriptive statistics in Section 8.1. Tests are run in R [127]. The actual tests are listed in Appendix K. The purpose of the tests is to confirm/reject the hypothesis:

H_0: There are no significant differences in cost and effectiveness when applying the testing strategies AT, RTP, and ATP on the complete model when combined with oracle O1.

Chapters 8 to 9 show results from comparing all the collected data for the three strategies when applied to the complete model and oracle O1. In order to make such a comparison, three statistical tests were executed: AT versus ATP, AT versus RTP, and ATP versus RTP. As multiple comparisons were made on the same sample, it would be sufficient to present the *p*-value. However, as Bonferroni correction is recommended by the literature [128], this kind of correction was used to avoid spurious positives when comparing the three strategies AT, RTP, and ATP. The original alpha $\alpha = 0.05$ was thus lowered to $\alpha = 0.05 / 3$ for the purpose of accounting for the number of comparisons.

Overall, ATP performed significantly better than both RTP and AT as to what mutation score regards. Accordingly, the cost when applying ATP was significantly higher than for RTP and AT.

Comparing AT with RTP, we see that for both measures on time the *p*-values are very low (p = 1.86e-09 for preparation time and p = 1.82e-06 for execution time). This means that the data is significantly different for the two strategies, even for $\alpha = 0.01$. When also looking at the effect size ($\hat{A}_{12} = 1$ for both preparation time and execution time), we can say that the probability of AT's preparation time and execution time being higher than for RTP is 100 percent. Interestingly, this does not regard the mutation score. There were no significant differences in the mutation scores achieved by AT as compared to RTP. The effect size of 0.5 shows that there is 50 percent probability for either strategy to perform better.

The figures for AT and ATP shows that ATP had the highest values for preparation time and execution time; the time spent on preparing and executing the test suites were significantly higher than for AT (p = 1.863e-09 for preparation time and p = 4.5e-06 for execution time). The large effect sizes indicate a 100 percent probability of higher preparation time and 91.4 percent probability of higher execution time. Again, no difference was found in mutation scores for ATP as compared to AT.

Furthermore, the differences in preparation time and execution time were significant also in the final comparison; RTP versus ATP (p = 1.863e-09 for preparation time and p = 1.863e-09 for execution time). The $\hat{A}_{12}$ statistics also show that RTP always required less time than ATP ($\hat{A}_{12} = 0$). Henceforth, the effect size is large.

This implies the following order: ATP costs more than AT which again costs more than RTP. Regardless of these differences, however, the three strategies performed similar in killing mutants. Thus, RTP can be considered being the most cost-effective strategy when using the complete model in combination with oracle O1.

Table 16 Paired Wilcoxon signed-rank test comparing strategies – preparation time

H_0	Oracle	Model	Measure	*p*-value	$\hat{A}_{12}$	Effect Size	Result	Sign. Diff. (CI)
AT = RTP	O1	Complete	Prep. time	1.86e-09	1	Large	AT > RTP	Yes (99%)
AT = ATP	O1	Complete	Prep. time	1.86e-09	1	Large	AT < ATP	Yes (99%)
RTP = ATP	O1	Complete	Prep. time	1.86e-09	0	Large	RTP < ATP	Yes (99%)

Table 17 Paired Wilcoxon signed-rank test comparing strategies – execution time

H_0	Oracle	Model	Measure	*p*-value	$\hat{A}_{12}$	Effect Size	Result	Sign. Diff. (CI)
AT = RTP	O1	Complete	Exec. time	1.82e-06	1	Large	AT>RTP	Yes (99%)
AT = ATP	O1	Complete	Exec. time	4.50e-06	0.086	Large	AT<ATP	Yes (99%)
RTP = ATP	O1	Complete	Exec. time	1.86e-09	0	Large	RTP<ATP	Yes (99%)

Table 18 Paired Wilcoxon signed-rank test comparing strategies – mutation score

H_0	Oracle	Model	Measure	*p*-value	$\hat{A}_{12}$	Effect Size	Result	Sign. Diff. (CI)
AT = RTP	O1	Complete	Mut. score	NA	O.5	No effect	AT = RTP	No
AT = ATP	O1	Complete	Mut. score	NA	O.5	No effect	AT = ATP	No
RTP = ATP	O1	Complete	Mut. score	NA	O.5	No effect	RTP = ATP	No

Based on the obtained results, the null hypothesis was rejected for the surrogate measures for cost. No evidence was found however to reject the null hypothesis with respect to effectiveness. Thus, there are significant differences in cost, but not in effectiveness when applying the testing strategies AT, RTP, and ATP when using oracle O1.

8.3 Cost-Effectiveness

This section focuses on the relation between cost and effectiveness. Each of the surrogate measures of cost and effectiveness is graphically illustrated by using colors in Figure 22. Dark colors indicate high values; light colors represent low values. We easily observe that five of the criteria provided good mutation score – LN2 had the worst results on mutation score. Looking at the cost for each strategy, ATP stands out with its high values for all measure on cost. RTP and LN3 perform quite similar. AT had the second highest cost, but similar effectiveness as ATP, RTP, and LN4.

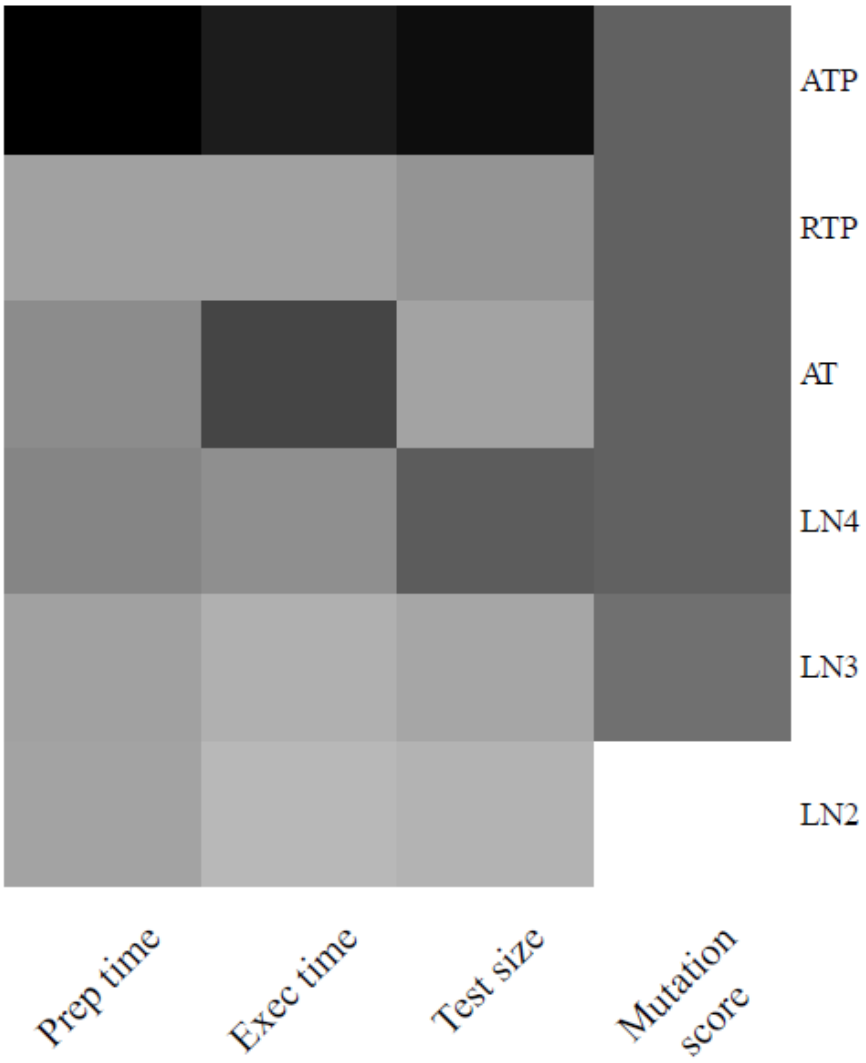


Figure 22 Preparation time, execution time, test-suite size and mutation score for each coverage criterion

Figure 23–Figure 26 show the relation between the surrogate measures of cost and effectiveness. In Figure 23–Figure 25 we see the number of killed mutants versus the mean preparation time, execution time, and total time for each strategy, respectively. Clearly, although killing all mutants, ATP is the most expensive criteria to prepare. On the other extreme, LN2 is very cheap; the fault-detection ability is accordingly poor. We observe from the four figures that the measures provide rather consistent results. There are, however, some differences in the results, e.g. preparation time versus execution time. Preparing the AT test suites took slightly more time than RTP but less time than LN4. Looking at the execution time on the other hand, Figure 23 shows that AT takes significantly more time than both RTP and LN4.

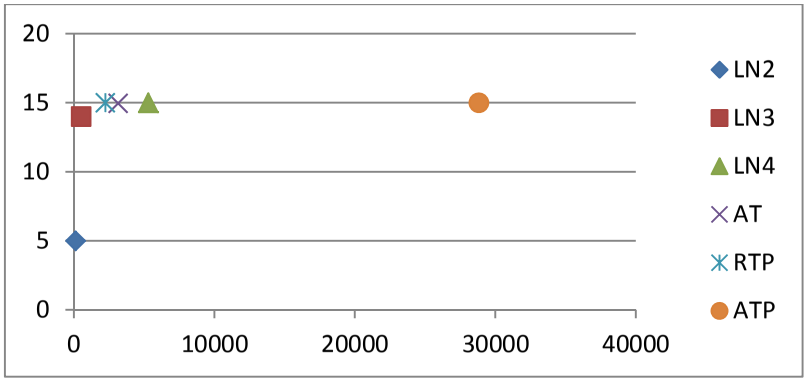


Figure 23 # mutants killed versus preparation time

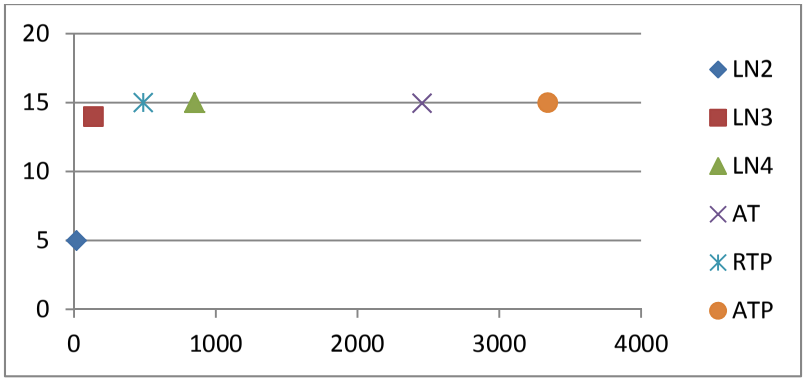


Figure 24 # mutants killed versus execution time

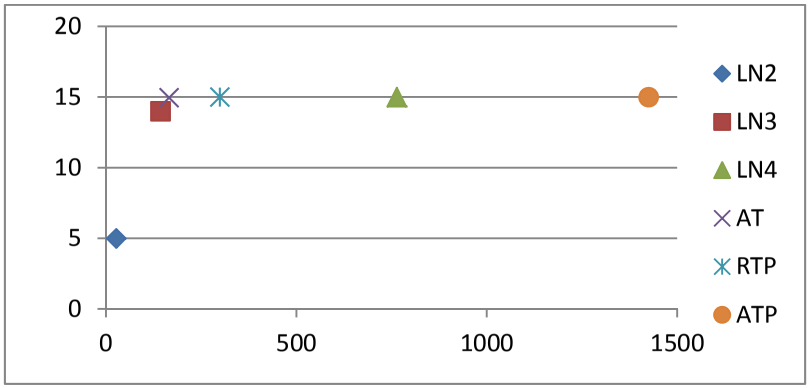


Figure 25 # mutants killed versus test-suite size

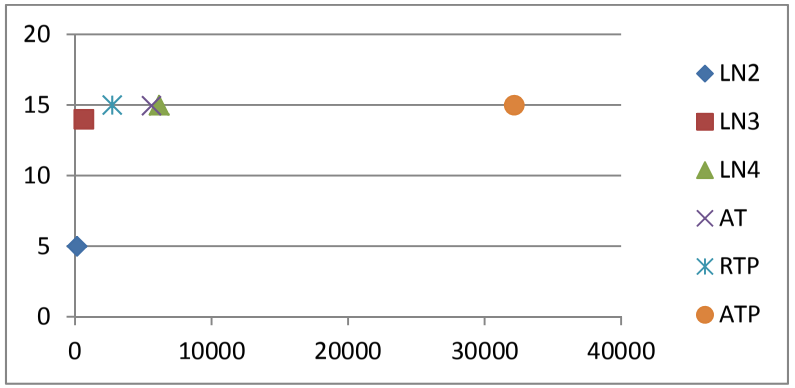


Figure 26 # mutants killed versus total time (prep + exec)

8.4 Analysis of Mutant Survival

We will now take a closer look at the nature of the mutants that were not killed by the applied test suites. The ATP, RTP, and LN4 test suites killed all mutants when executing each of the 30 test suites.

On average, all mutants were killed by the AT test suites. Investigations of the individual test suites show that only one of the 30 test suites (AT11) did not kill mutant M15. Recall that M15 was a fault that consisted of adding a guard to a transition. The guard evaluates the particular speed mode of the robot; the speed had to be `Extra Slow` to enable transitioning. In the correct version, firing of the transition should take place regardless of the speed mode. Explanations for why this mutant was not killed may be that the generated oracle provided test data for the other variables in the guard that prevented the transitioning. The remaining 29 test suites killed all of the 15 mutants. Neither LN3 killed M15. This, however, has a logical explanation in that LN3 does not reach that particular state.

LN2 performed the worst: ten mutants were not killed (only M1, M2, M12, M13, and M14 were killed).

8.5 Related Work

Offutt and Abdurazik [15] addressed system level testing by generating test cases from UML state machines. In an empirical study, they demonstrated two techniques and evaluated their fault-detection ability, of which one of the techniques was ATP. The Cruise Control system, developed for research purposes, was seeded with 25 faults and tested running 34 ATP tests generated from a proof-of-concept test data generation tool. Four of the seeded faults were actual faults, detected during the initial implementation. Results showed that ATP killed 72 percent (18/25).

Offutt *et al.* [16] continued the investigations on state-based testing to see if the specification-based testing criteria could be practically applied. They evaluated the efficiency of state-based test criteria in terms of fault-detection ability and obtained branch coverage. Among other criteria, AT and ATP were applied in a case study and compared with respect to fault-detection ability and branch coverage. A modified version of the Cruise Control system (400 LOC of C) was mutated using 24 faults (of which 4 were naturally occurring faults). Obtained results showed that the weakest coverage, AT (12 test cases), detected 15/24 faults, and ATP (34 test cases) detected 18/24 faults.

Among several other criteria, the ATP coverage criterion was compared to mutation based criteria by Paradkar [82]. The study reported that mutation-based testing had higher fault-detection effectiveness, but at a higher cost than the structured criteria. By running 39 ATP test cases on the Java implementation (389 LOC) of an ATM application, 89/166 (54 percent) mutants were killed. The mutants were manually generated by seeding, randomly, faults according to mutation operators defined in [130]. In addition, they applied Control Flow Disruption (CFD), and Scalar Variable Replacement (SVR).

Chevalley and Thévenod-Fosse [78] found a weakness of AT in that a fault could possibly not be triggered by the particular test case input value for a particular transition. For that reason, the weakness in the criterion was compensated by exercising each transition several times. The results were based on 1,559 mutants of an avionics system (6,500 LOC); a mutation score of 91 percent was reached (1,423/1,559 mutants were killed). The Flight Guidance System was a research version provided by the Advanced Technology Center of Rockwell-Collins.

Briand *et al.* [22] empirically investigated the cost and fault-detection effectiveness for the most referenced coverage criteria based on UML statecharts: AT, ATP, and FP [15], and a modified version of the RTP coverage referred to as transition tree (TT) [2, 31]. Three case studies were used in the evaluation: (1) a container class from an academic example program, (2) the Cruise Control system, and (3) an implementation of a video recorder. The two former studies were real-time systems. Artificial mutation operators were used to create 101, 91, and 139 mutants respectively for the three cases. The following conclusions were drawn from the study: (1) AT did not provide an adequate level of fault detection, (2) ATP detected nearly all faults, but not without an enormous increase in cost compared to AT, (3) TT was evaluated to be more cost-effective than AT and ATP, although the result depended on two factors: the extent to which guard conditions were present in the statechart, and the extent to which the transition tree captured realistic and meaningful usage scenarios.

In a series of three controlled experiments, Briand *et al.* [12] evaluated two variants of RTP testing, and CP testing, in terms of cost-effectiveness, and proposed a way to combine them. Students were used as subjects. The following programs were used in the experiments: (1) a container class from an academic software system, (2) a container class from a real DNS system, and (3) two control classes from a real DNS system. Two different oracle strategies were compared; the state-invariant oracle versus the precise oracle. Artificial mutation operators were applied in order to evaluate the cost-effectiveness. In the three experiments, 24, 42, and 81 mutants were generated. Results showed that RTP testing is not likely to be sufficient in most situations as significant numbers of faults remained undetected (from 10 percent to 34 percent) on average across subject classes. This is especially true when a weaker form of round-trip was used where only one of the disjuncts in guard conditions was exercised. By combining RTP with CP testing, however, a large percentage of latent faults could potentially be detected by CP after SBT was applied, yet at significant increase in cost, implying that selection of subsets may be necessary.

The effectiveness of the RTP strategy was later investigated by Antoniol *et al.* [80] in a case study. The RTP strategy was applied in a C++ example program consisting of 450 LOC (two classes and 45 methods). The main class under test was a container class. Artificial mutation operators were used to seed 44 faults covering 8 mutation operators. The study concluded that the RTP strategy is reasonably effective at detecting faults; 88 percent of the faults were detected as compared to 69 percent for random testing. Moreover, their results showed that RTP left certain types of faults undetected, and suggested that by augmenting

RTP with category-partition (CP) testing, the fault-detection can be enhanced, although at an increase in cost that must be taken into account.

A study by Briand *et al.* [81] was conducted that aimed at investigating how data flow information could be used to improve the cost-effectiveness of state-based coverage criteria when more than one tree existed. Two case studies were carried out: (1) the Cruise Control system was inserted with 91 faults using artificial mutation operators, and (2) 131 mutants were generated from an implementation of a video recorder. Results showed that data flow information was useful for selection of the best cost-effective transition tree.

Mouchawrab *et al.* [85] addressed the impact of using statecharts for testing class clusters that exhibit a state-dependent behavior, and reported on a controlled experiment that investigated the effectiveness of SBT using RTP when compared and combined to white-box, structural testing. The experiment involved 48 students who were assigned to generate tests for the `OrdSet` example class, and the Cruise Control system using RTP, and block and edges coverage. No differences in the fault-detection effectiveness of the two strategies were found by executing the generated tests on 624 mutants of the `OrdSet` class and 386 mutants of the Cruise Control system. Results from applying RTP showed an average mutation score of approximately 79 percent in the `OrdSet` case and as low as 22 percent in the Cruise Control case. Combining the strategies, however, proved to be significantly more effective. Number of killed mutants (i.e., killed by RTP and the structural criterion) varied from 80–100 percent in the OrdSet case and 61–100 percent in the Cruise Control case. The fault detection effectiveness was found to vary to a large extent depending on how precisely the statechart described the behavior of the software under test.

Continuing the investigations of the impact of state-machine testing (the round-trip path coverage criterion in particular) on fault detection and cost when compared with structural testing, Mouchawrab *et al.* [92] conducted four controlled experiments. Results showed that there was no significant difference between the two strategies regarding fault-detection effectiveness. Combining the two strategies, however, yielded significantly more effective results. The number of mutants varied from 382 to 1,176.

The round-trip path criterion was further studied by Briand *et al.* [98] in the context of UML state machines with focus on how to improve the criterion's fault-detection effectiveness. They investigated how data flow analysis on OCL guard conditions and operation contracts could be used to "further refine the selection of a cost-effective test suite among alternative, adequate test suites for a given state machine criterion" [98]. A methodology on how to perform data flow analysis of UML state machines was presented.

Results from two case studies, a VCR system (1,000 LOC) and the well-known Cruise Control system (460 LOC), suggested that data flow information in a transition tree could be used to select the tree with the highest fault-detection ability. Artificial mutation operators were used to generate 131 and 91 mutants for VCR and CC, respectively. For VCR, the number of killed mutants varied from 71–76 percent among the 12 trees generated, whereas 85, 91, and 96 percent of the mutants were killed by the three transition trees generated from the CC.

In [99], Khalil and Labiche addressed the assumptions about the round-trip path strategy regarding the equivalency of exercising paths in the tree that do not always trigger complete round trip path versus covering round-trip paths. They investigated the consequences of the assumption not being held in practice. Finally, they proposed yet a new algorithm for generating the transition tree, which resulted in higher efficiency and lower cost. They created 187 mutants for the Cruise Control example and 417 faulty versions of the OrderedSet example. Cost was measured by several surrogate measures: the number of test cases, the number of states in trees, and the number of events in trees.

8.6 Discussion

This section compares the observed results to related work as just described in Section 8.5. Please recall that in this study, a hierarchical orthogonal module in a control system was applied in the evaluation of six state-based coverage criteria. Twenty-six mutated versions of the SUT were generated by seeding real faults that were extracted from a global field study conducted in three of ABB's departments. Only fifteen of these faults were detectable by conformance testing, as the remaining eleven faults were sneak paths.

There are several factors that distinguish the abovementioned studies from the study presented in this thesis. First of all, the majority of existing studies on cost-effectiveness related to SBT, except for [78], utilize small non-industrial or example cases which are significantly smaller or less complex than the SUT. The number of tests in the generated test suites and LOC reflect that difference. For example, recall that the number of tests generated for ATP in this study was 1,425 as compared to 34 in [15] and [16]. In particular, recall the frequent use of the Cruise Control case in existing research. Second, the applied oracle is most often not specified; the lack of oracle information makes it difficult to provide meaningful comparisons of the results. Only Briand *et al.* [12] specifically addressed and, in fact, compared the type of oracles used. Third, the number of seeded mutants is in most cases

higher than in this study, but then again, (forth) the seeded faults are primarily artificial in existing studies. Nevertheless, the latter difference is interesting due to the lack of studies on artificial versus real faults. Fifth, the level of details in the test model is insufficiently specified. Sixth, there are no data on cost other than test-suite sizes.

Now, by looking at the reported results on fault-detection ability, we see that ATP killed more mutants (100 percent) in this study as compared to [15], [16], and [82]. The latter reported a mutation score as low as 54 percent. This may of course be explained by the differences in the studies as previously stated. The relationship between the fault-detection ability of AT versus ATP presented in [16], however, seem to be rather consistent to results in this study.

As we can see from the existing studies, results show great variations in the fault-detection ability – from 63 to 91 percent. In spite of the differences between the study of Briand *et al.* [22] and this study, results reported in this study support the findings of Briand *et al.*; except from the findings regarding AT not providing an adequate level of fault detection. In this study, by using a complete test model and oracle O1 (checking state invariant and state pointer), we saw that AT detected as many mutants as ATP. Then again, the study of Briand *et al.* involved many more mutants albeit artificial.

Empirical studies on RTP have shown that, in terms of cost and effectiveness, this particular criterion is a compromise between the weak AT and the more expensive ATP criteria if the selected transition trees were the most effective in cases where several trees existed [22]. The results in this study support these findings. Note, however, that also for RTP, reported results in existing studies are highly variable (22–90 percent).

8.7 Summary

This chapter concerned cost-effectiveness of the state-based coverage criteria all transitions (AT), all round-trip paths (RTP), all transition pairs (ATP), paths of length 2 (LN2), paths of length 3 (LN3) and paths of length 4 (LN4) when applied to a detailed state-based test model using oracle O1.

Descriptive statistics for the obtained results on cost and effectiveness in Section 8.1 showed that LN2 (27 test cases) provided the smallest test suite; ATP (1,425 test cases) had the largest test suite. The median among the test-suite sizes was 232.4 test cases.

Looking at the mean values, we observe that LN2 had the lowest values for preparation time (126 seconds). LN3 had the second lowest value, 509 seconds. RTP ranked third with

531 seconds. A large increase, almost 87 percent, was seen between preparation time for RTP and AT (3,995 seconds). The second highest measure collected for preparation time was LN4 with 5,295 seconds. The highest value, 28,819 seconds, was observed for ATP.

Results show for execution time that, again, LN2 provided the lowest value (18 seconds). LN3 was measured to 136 seconds, followed by RTP with 489 seconds. Almost doubling the time seen for RTP, LN4 was measured to use 850 seconds on executing the test suite. The second highest time was measured for AT – 2,455 seconds. Finally, execution of the ATP test suite took 3,341 seconds.

Considering the mutation score results ranked by mean from low to high, the summary showed that LN2 performed significantly poorer than the other coverage criteria (5/15 killed mutants). A large gap was to be found between LN2 and the next result; LN3 with 0.933 (14/15 killed mutants). Quite similar, AT resulted in a high mutation score mean, 0.997. The best mutation score mean came with RTP, ATP, and LN4 – all mutants were killed.

Section 8.2 regarded statistical tests. The paired Wilcoxon signed-rank test was applied to the replicated data, i.e., for AT, RTP, and ATP. All tests executed on preparation and execution time resulted in significantly different results – ATP spent significantly more time on both preparation and execution of the test suites than AT and RTP. No significant differences were found in data collected on mutation score. Both RTP and ATP killed all mutants. AT killed all mutants in 29 of 30 test suites – the final test suite killed 14 out of 15 mutants.

Conclusions: The results indicate that LN2 might be too weak as a testing strategy. The other testing strategies performed similar with respect to mutation score, but with varying costs – ATP was the most expensive criterion. Having rather similar cost-effectiveness, LN3 and RPT were suggested by the results as the most cost-effective strategies.

9 Case Study 2 – How does Varying the Oracle Affect the Cost-Effectiveness?

Whereas the previous chapter compared six coverage criteria, executed on the complete model using one type of oracle, the case study presented in this chapter aims to answer the second research question:

RQ2: How does varying the oracle affect the cost-effectiveness?

Results are based on running tests when applying a modified version of the oracle used in Chapter 8. Please recall from Section 7.1 the difference between the two oracles O1 and O2. The former checks both the state invariant in addition to the pointer to the current system state, whereas the latter only checks the current system state.

The answer to this question will be provided by investigating the influence of the two oracles on the cost-effectiveness of the six coverage criteria in focus of this thesis. Section 9.1 presents descriptive statistics, whereas Section 9.2 provides statistical tests. Descriptive statistics from Section 8.1 are restated in this Chapter for the purpose of making comparisons between the two oracles. Section 9.3 regards cost versus effectiveness. Undetected mutants are addressed in Section 9.4. Obtained results are discussed towards existing research in Section 9.5. Furthermore, Section 9.6 discusses results from related work against results from this study. Lastly, the obtained results are summarized in Section 9.7. The data material can be found in Appendix G (regarding oracle O1) and Appendix H (regarding oracle O2).

9.1 Descriptive Statistics

9.1.1 Descriptive Statistics for Cost

9.1.1.1 Test-Suite Size

As the choice of oracle does not affect test-suite sizes, this particular measure is irrelevant for the second research question and therefore excluded. Sizes are determined by the testing strategies and the system specification, more precisely the number of transitions in the state-machine diagram. Hence, changing the oracle will not have impact on this surrogate measure for cost. More importantly, however, is the oracle's influence on cost measured as time.

9.1.1.2 Time

Table 19 and Table 20 show the descriptive statistics for preparation and execution time, respectively. Recall that time was measured in seconds.

From Table 19 we see that three of the six testing strategies spent slightly more time on preparing the test suites when combined with oracle O2 than with oracle O1. This regards LN3, RTP, and LN4. No difference between the oracles regarding preparation time was seen for LN2. Two strategies spent more time on preparing the test suites when applying oracle O1 as compared to O2, namely AT and ATP.

Table 19 Descriptive statistics for preparation time when applying oracle O1 and O2

Strategy	Model	Oracle	Min (sec)	Q1 (sec)	Mean (sec)	Median (sec)	Q3 (sec)	Max (sec)	St Dev (sec)	N	Diff O1 by O2 (%)
LN2	Complete	O1	126	126	126	126	126	126	-	1	0.0
		O2	126	126	126	126	126	126	-	1	
LN3	Complete	O1	509	509	509	509	509	509	-	1	-1.4
		O2	516	516	516	516	516	516	-	1	
RTP	Complete	O1	484	512	531	525	545	607	28	30	-0.2
		O2	476	492	533	523	559	675	51	30	
LN4	Complete	O1	5,295	5,295	5,295	5,295	5,295	5,295	-	1	-3.8
		O2	5,494	5,494	5,494	5,494	5,494	5,494	-	1	
AT	Complete	O1	2,506	2,763	3,995	3,013	3,665	15,939	3,264	30	7.5
		O2	2,249	2,474	3,715	2,766	3,394	15,663	3,280	30	
ATP	Complete	O1	28,377	28,576	28,819	28,797	29,028	29,398	273	30	0.6
		O2	28,305	28,409	28,641	28,504	28,787	29,548	345	30	

Results for execution time show that there were significant cost reductions by using oracle O2. This regards all six testing strategies. The greatest improvement, however, was seen for AT closely followed by ATP.

Table 20 Descriptive statistics for execution time when applying oracle O1 and O2

Strategy	Model	Oracle	Min (sec)	Q1 (sec)	Mean (sec)	Median (sec)	Q3 (sec)	Max (sec)	St Dev (sec)	N	Diff O1 by O2 (%)
LN2	Complete	O1	18	18	18	18	18	18	-	1	260.0
		O2	5	5	5	5	5	5	-	1	
LN3	Complete	O1	136	136	136	136	136	136	-	1	385.7
		O2	28	28	28	28	28	28	-	1	
RTP	Complete	O1	341	460	489	504	524	607	54	30	412.5
		O2	80	88	95	97	101	119	9	30	
LN4	Complete	O1	850	850	850	850	850	850	-	1	391.3
		O2	173	173	173	173	173	173	-	1	
AT	Complete	O1	1,765	2,108	2,455	2,377	2,785	3,617	469	30	491.0
		O2	293	358	415	417	473	571	71	30	
ATP	Complete	O1	2,607	2,972	3,341	3,381	3,669	3,978	429	30	487.3
		O2	429	493	569	547	654	773	101	30	

The means from Table 19 and Table 20 are graphically visualized in Figure 27. The figure shows consistent results between the oracles what comes to preparation time. The execution times, however, were significantly reduced when applying oracle O2.

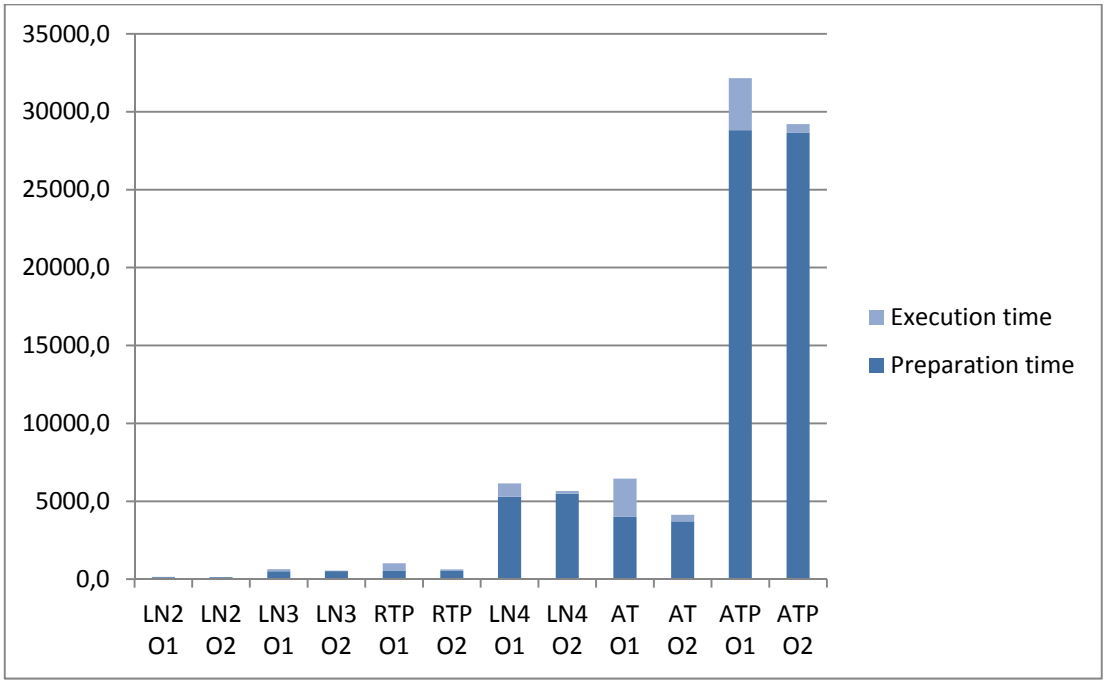


Figure 27 Preparation and execution time in seconds

The standard deviations displayed in Table 19 and Table 20 illustrate the variations in results among the generated trees within the strategies RTP, AT, and ATP. This will be further addressed in the following paragraphs.

Comparing Means

The histograms and box plots in Figure 28 and Figure 29 show the distribution of the time spent on preparing and executing the test suites, respectively.

Figure 28 presents the performance of AT, RTP, and ATP regarding preparation time. Again, none of the histograms show a normal distribution. The distributions are skewed to the left.

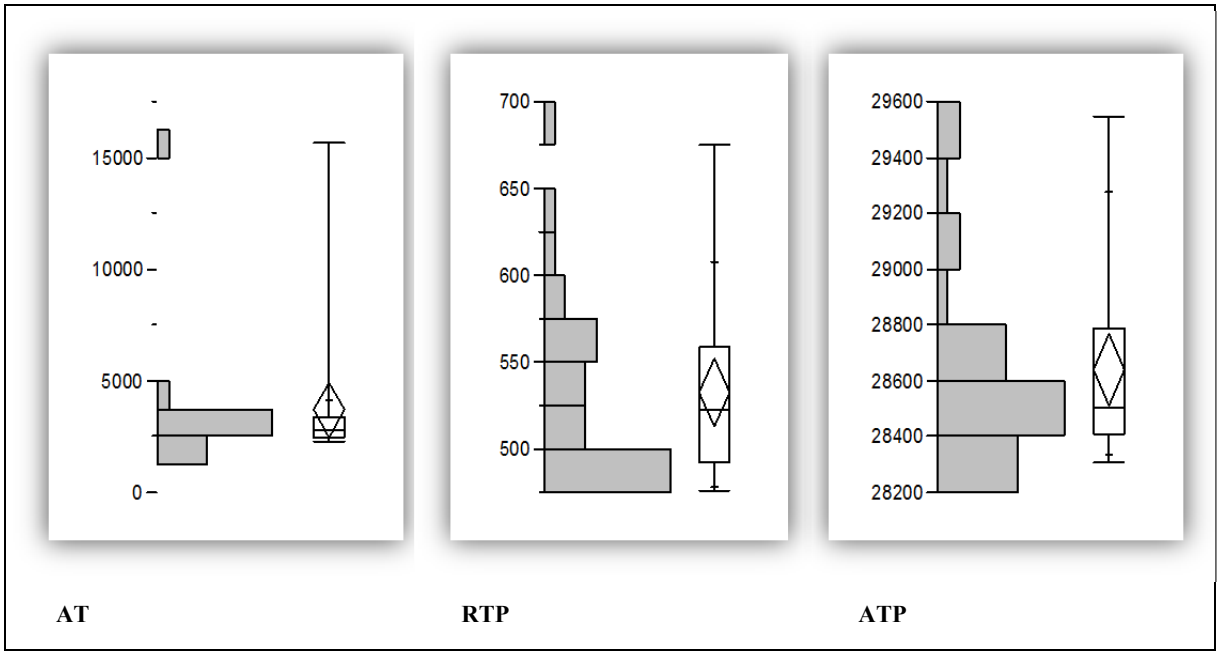


Figure 28 Distribution – preparation time – complete O2

The distribution of AT, is centered around 3,715 seconds. Investigations of the observations and by looking at the IQR from Table 21, we see that there are two outliers around 15,600 seconds – whom have a dramatic impact on the standard deviation. Also, the mean of the distribution significantly differs from the median. This is cause by the presence of the two outliers, however.

Next, there were 16 observations within the lower and upper quartiles of RTP. The mean was 533 seconds. In the upper range, we see one outlier at 675 seconds.

Finally, ATP's distribution has its mean at 28,641 seconds. The observations are located close to the mean. Yet, there are two outliers among the observations in the upper range around 29,500 seconds.

This indicate a wide spread in data for AT, whereas RTP and ATP have more consistent observations. After removing the two outliers in the AT observations, however, we still see a distribution with a high standard deviation.

Table 21 Inter quartile ranges – preparation time

Strategy	Q1	Q3	Q3-Q1	IQR	Lower limit	Upper limit
AT	2,474	3,394	920	1,380	1,095	4,774
RTP	492	559	67	100	392	658
ATP	28,409	28,787	378	567	27,842	29,354

Figure 29 displays the spread in the collected data on execution time. The execution time for AT varies from 293 seconds to 571 seconds, for RTP we see a range from 80 seconds to 119 seconds, and for ATP the time is spread from 429 seconds to 773 seconds.

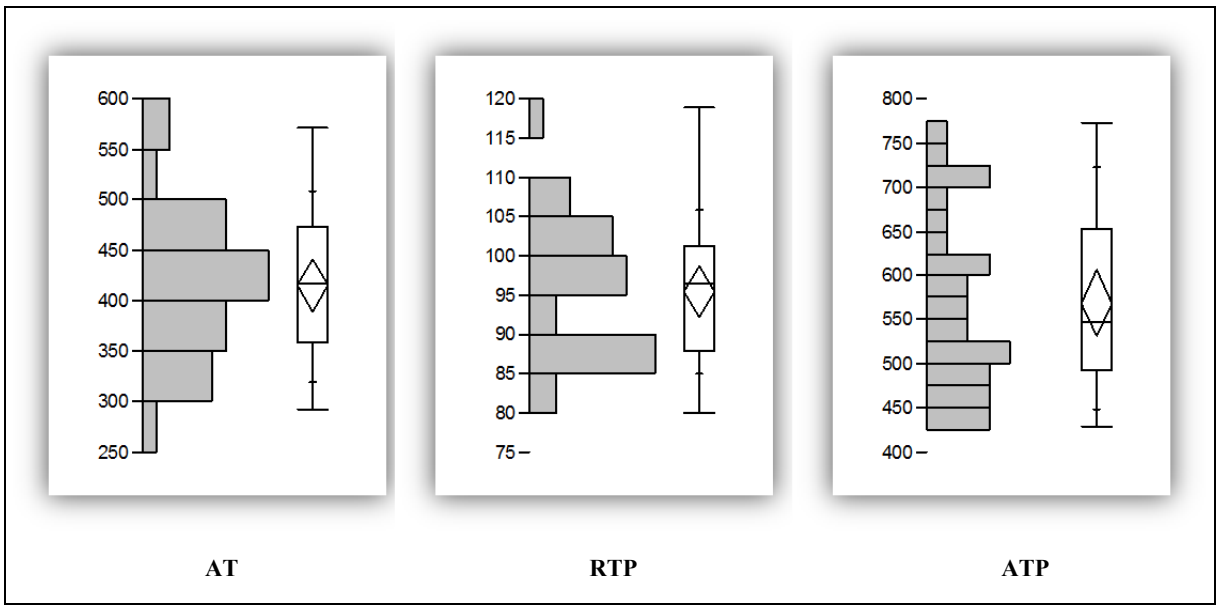


Figure 29 Distribution – execution time – complete O2

AT's distribution is close to normally distributed. The minimum value among the observations was 293 seconds, the mean was 416 seconds, and the maximum value was 571 seconds. No outliers were observed.

The RTP distribution is spread from 80 to 119 seconds, having the mean at 95 seconds. Nor this distribution had any outliers among the observations.

Finally, in the distribution of ATP, we see three peaks around 506, 614, and 717 seconds. The mean was at 569 seconds, the minimum at 429 seconds, and the maximum at 773 seconds. No outliers were observed for this strategy.

Table 22 Inter quartile ranges – execution time

Strategy	Q1	Q3	Q3-Q1	IQR	Lower limit	Upper limit
AT	358	473	115	173	185	645
RTP	88	101	14	20	68	122
ATP	493	654	161	242	251	895

9.1.2 Descriptive Statistics for Effectiveness

This section describes main features of the collected data on effect, which regards mutation score – the surrogate measure of effectiveness.

Table 23 Descriptive statistics for mutation score

Strategy	Model	Oracle	Min	Q1	Mean	Median	Q3	Max	Std Dev	N	Diff O1 by O2 (%)
LN2	complete	O1	0.33	0.33	0.33	0.33	0.33	0.33	0.00	1	149.8
		O2	0.13	0.13	0.13	0.13	0.13	0.13	-	1	
ATP	complete	O1	1.00	1.00	1.00	1.00	1.00	1.00	0.00	30	36.9
		O2	0.60	0.73	0.73	0.73	0.73	0.80	0.03	30	
LN3	complete	O1	0.93	0.93	0.93	0.93	0.93	0.93	0.00	1	27.2
		O2	0.73	0.73	0.73	0.73	0.73	0.73	-	1	
AT	complete	O1	0.90	1.00	1.00	1.00	1.00	1.00	0.02	30	25.3
		O2	0.73	0.80	0.80	0.80	0.80	0.80	0.02	30	
RTP	complete	O1	1.00	1.00	1.00	1.00	1.00	1.00	0.00	30	25.0
		O2	0.80	0.80	0.80	0.80	0.80	0.80	0.00	30	
LN4	complete	O1	1.00	1.00	1.00	1.00	1.00	1.00	0.00	1	25.0
		O2	0.80	0.80	0.80	0.80	0.80	0.80	-	1	

Table 23 displays the results for both oracles for each testing strategy. For all strategies, oracle O1 obtained higher mutation score means than oracle O2. The greatest difference was found when using LN2, followed by ATP. Rather similar differences were seen for LN3, AT, RTP, and LN4.

The mutation-score means obtained by applying O1 and O2, displayed in Table 23, are graphically visualized in Figure 30. We see that the complete test model combined with oracle O1 resulted in an overall higher mutation score than by combining the complete test model with oracle O2.

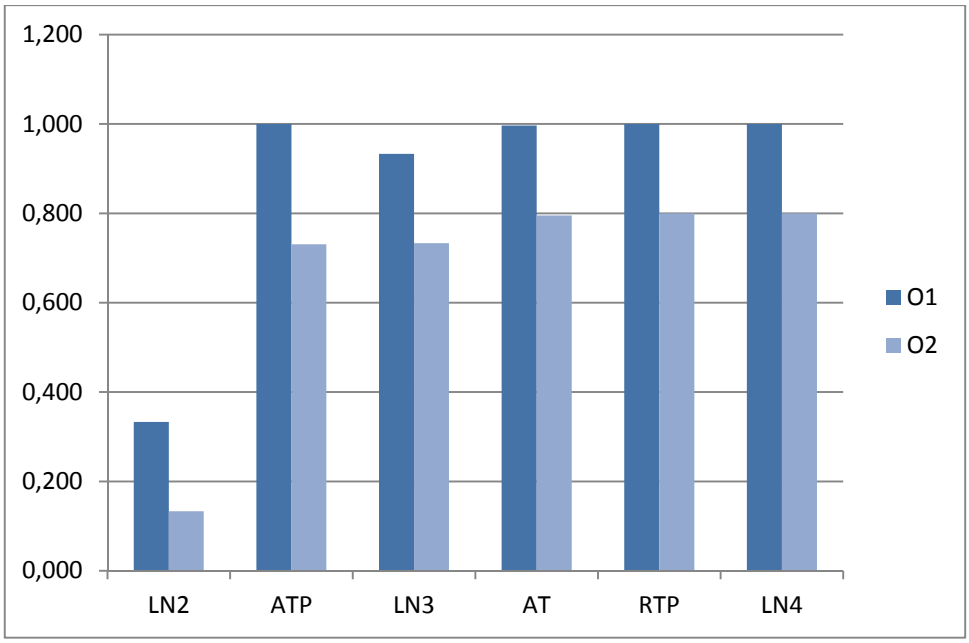


Figure 30 Mutation score means

Comparing Means

This section investigates the replications of AT, RTP, and ATP for the purpose of identifying possible statistical variability in the collected data on mutation scores. Also for this set of observations we hardly see any deviation from the mean (see Table 23). The histograms in Figure 31 show quite consistent distributions of the mutation scores.

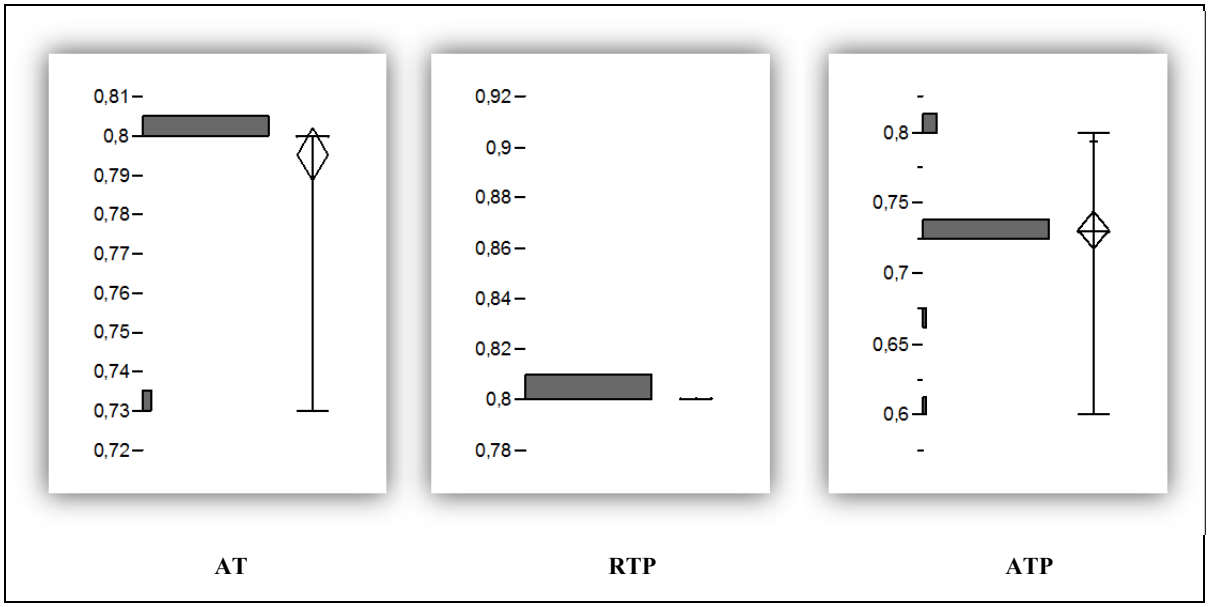


Figure 31 Distributions mutation score complete O2

From the histogram of AT's distribution, we see that 28 test suites achieved 0.8 as the mutation score (killed 12/15 mutants) – the remaining two test suites only killed 11/15

mutants, and hence obtained a mutation score of 0.73). The 30 generated RTP test suites all provided the same mutation score, 0.8. The last distribution, ATP, had some spread in the results: three test suites killed 12/15 mutants (mutation score: 0.8), 25 test suites killed 11/15 mutants (mutation score: 0.73), whereas the final two test suites only killed 10/15 (mutation score: 0.6) and 9/15 mutants (mutation score: 0.6).

Table 24 Inter quartile ranges – mutation time

Strategy	Q1	Q3	Q3-Q1	IQR	Lower limit	Upper limit
AT	0.80	0.80	0.00	0.00	0.80	0.80
RTP	0.80	0.80	0.00	0.00	0.80	0.80
ATP	0.73	0.73	0.00	0.00	0.73	0.73

This means that there is little variation present in the collected data on mutation score for each testing strategy.

The replicated data will be further tested for statistical significance between the three strategies in the next section.

9.2 Statistical Tests

This section provides statistical tests on the data described by descriptive statistics in Section 9.1. The paired Wilcoxon signed-rank test with a 0.99 confidence level was executed on the collected data on preparation time, execution time and mutation score when applying two different oracles on the complete model. Tests were run in R [127]. The actual tests are listed in Appendix L. The purpose of the tests is to confirm/reject the hypothesis:

H_0: There are no significant differences in cost and effectiveness when applying two different oracles for each of AT, RTP, and ATP on the complete model.

Let us look at Table 25 for the Wilcoxon tests on preparation time. Results show a low *p*-value (1.82e-06) for AT, indicating that there was a difference between the collected preparation time for oracle O1 and O2.

The difference is only significant, however, at a 95 percent level. Furthermore, the results show a medium effect size; the probability that a preparation time is higher when applying O1 as compared to O2 is 67 percent. For RTP on the other hand, results cannot be said to be significantly different (p = 1). Preparation times for O1 and O2 in combination with ATP were significantly different (p = 0.033) at a 99 percent CI. The $\hat{A}_{12}$ value tells us that there should be a 72 percent probability of having a higher preparation time when applying oracle O1 compared to O2.

Table 25 Paired Wilcoxon signed-rank test comparing oracles O1 and O2 – preparation time

H_0	Strategy	Model	Measure	p-value	$\hat{A}_{12}$	Effect Size	95 % CI for $\hat{A}_{12}$	99 % CI for $\hat{A}_{12}$	Result	Sign. Diff. (CI)
O1 = O2	AT	Complete	Prep. time	1.82e-06	0.67	Medium	[0.521, 0.794]	[0.474, 0.824]	O1>O2	Yes (95%)
O1 = O2	RTP	Complete	Prep. time	1	-	-	[NA, NA]	[NA, NA]	O1=O2	No
O1 = O2	ATP	Complete	Prep. time	0.033	0.72	Large	[0.573, 0.835]	[0.524, 0.860]	O1>O2	Yes (99%)

Table 26 displays the test results for execution time, comparing oracle O1 and O2. Significant differences at a 99 percent level were found for AT (p = 1.82e-06), RTP (p = 1.82e-06), and ATP (p = 1.86e-09). The effect size is large; in fact there is a 100 percent chance that the execution time for oracle O1 is larger than for oracle O2.

Table 26 Paired Wilcoxon signed-rank test comparing oracles O1 and O2 – execution time

H_0	Strategy	Model	Measure	p-value	$\hat{A}_{12}$	Effect Size	Result	Sign. Diff. (CI)
O1 = O2	AT	Complete	Exec. time	1.82e-06	1	Large	O1>O2	Yes (99%)
O1 = O2	RTP	Complete	Exec. time	1.82e-06	1	Large	O1>O2	Yes (99%)
O1 = O2	ATP	Complete	Exec. time	1.86e-09	1	Large	O1>O2	Yes (99%)

Moreover, Table 27 shows that there is a 100 percent chance that O1 kills more mutants than O2. The difference is significant at a 99 percent level.

Table 27 Paired Wilcoxon signed-rank test comparing oracles O1 and O2 – mutation score

H_0	Strategy	Model	Measure	*p*-value	$\hat{A}_{12}$	Effect Size	Result	Sign. Diff. (CI)
O1 = O2	AT	Complete	Mut. Score	1.08e-07	1	Large	O1>O2	Yes (99%)
O1 = O2	RPT	Complete	Mut. Score	4.61e-08	1	Large	O1>O2	Yes (99%)
O1 = O2	ATP	Complete	Mut. Score	2.76e-07	1	Large	O1>O2	Yes (99%)

As we can see from the statistical test results, the null hypothesis was rejected for all strategies except from RTP when considering the preparation time. That is, with the exception of the preparation time for RTP, there are significant differences in cost and effectiveness when applying two different oracles for each of AT, RTP and ATP on the complete model.

9.3 Cost-Effectiveness – Oracle O1 versus Oracle O2

This section regards the relation between cost and effectiveness for test suites combined with each of the two oracles. Figure 32 graphically illustrates the relationship between cost and effectiveness by displaying the surrogate measures for cost, test-suite size and time (preparation and execution time), and the surrogate measure for effectiveness, namely mutation score. Obviously, an ideal situation would have been low values for cost, yet high mutation scores. This is not, however, the case for all combinations of coverage criterion and oracle. In particular, we observe from Figure 32 that ATP combined with both oracles O1 and O2 is positioned far away from the desired area. The results suggest that the mutation score is negatively affected by all coverage criteria when using oracle O2.

One assumption to make is that a stronger coverage criterion should have less need for a strong oracle as compared to weaker coverage criteria having less code exercised. As we see from Figure 32, a strong oracle combined with a weaker coverage criterion clearly improves the fault-detection ability.

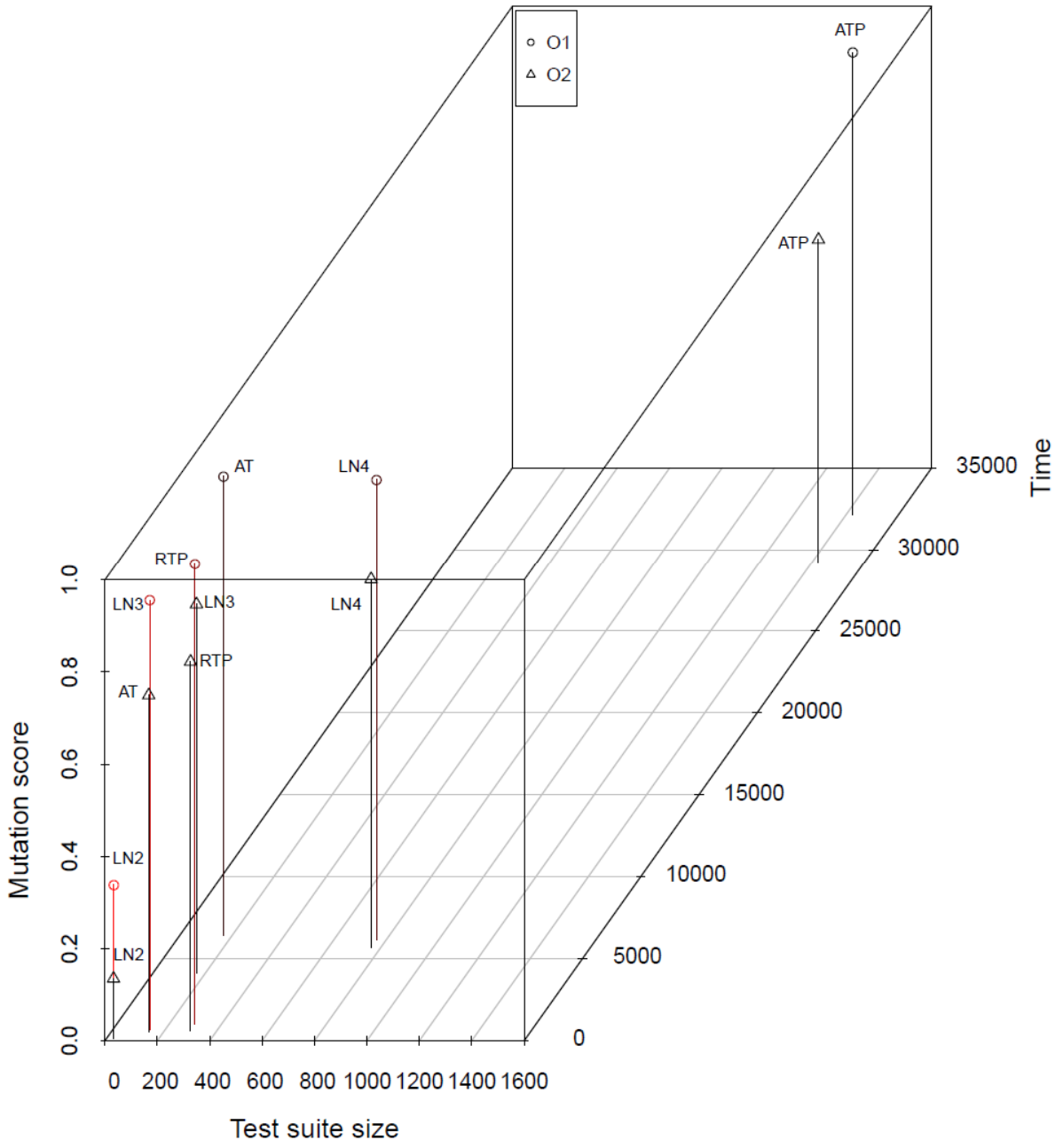


Figure 32 Oracle O1 versus oracle O2 – mutation score, test-suite size and time

Figure 33–Figure 36 present the number of killed mutants over each of the surrogate measure for cost. Even though the two oracles achieved rather similar cost-effectiveness when cost is measured in preparation time, we observe that using oracle O1 resulted in an overall higher cost-effectiveness as compared to oracle O2. The highest impact was seen for LN2, followed by LN3. Larger differences were seen for cost-effectiveness when focusing on execution time; oracle O2 achieved higher cost-effectiveness than oracle O1. When using test-

suite size as the surrogate measure of cost, we see that O1 obtained the highest cost-effectiveness.

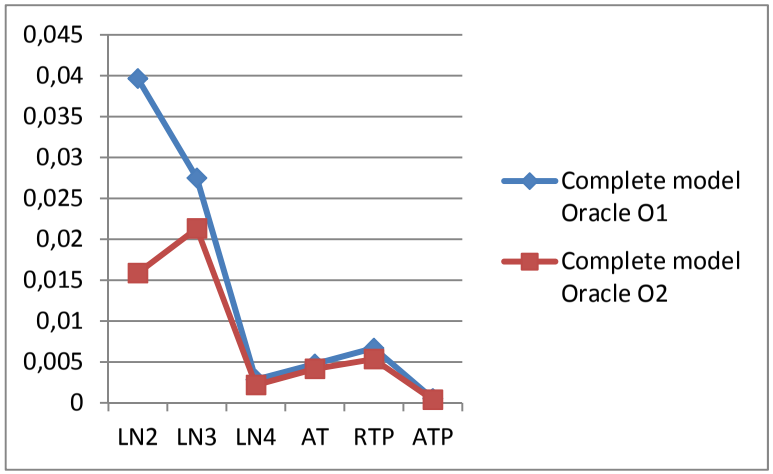


Figure 33 Number of mutants killed over preparation time

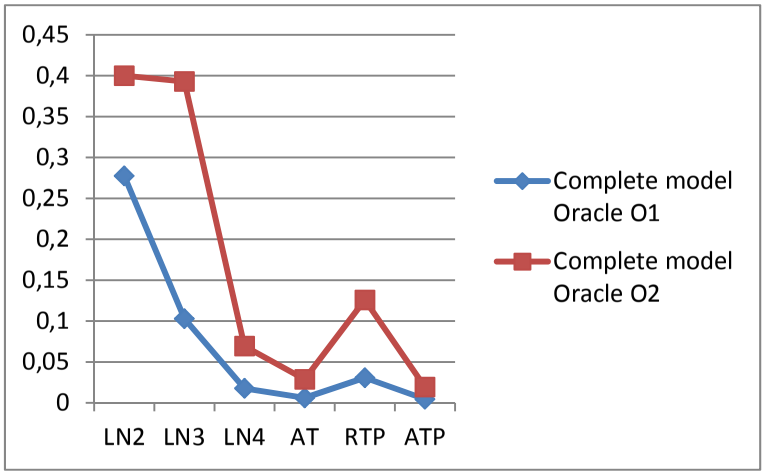


Figure 34 Number of mutants killed over execution time

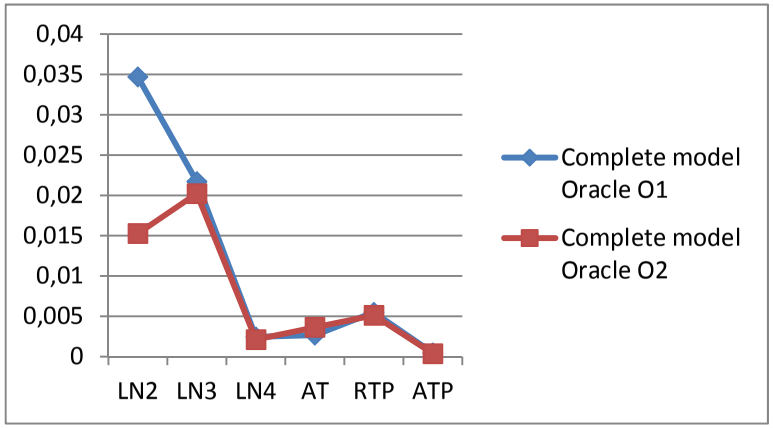


Figure 35 Number of mutants killed over total time

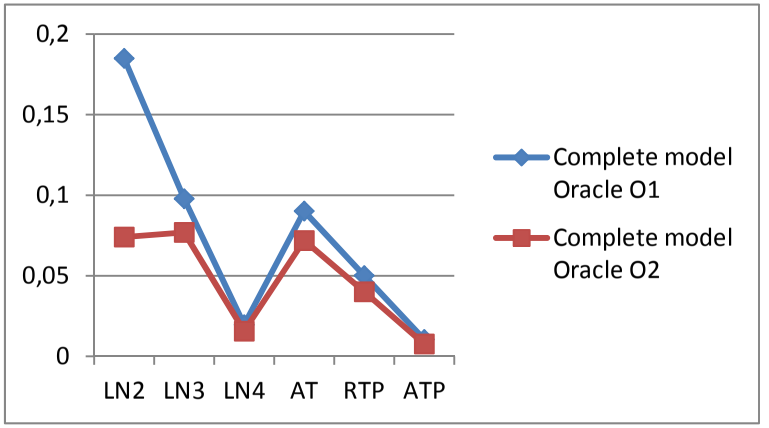


Figure 36 Number of mutants killed over test-suite sizes

9.4 Analysis of Mutant Survival

This section seeks to identify which types of mutants could not be killed by the stronger oracle O1 as compared to the weaker oracle O2.

Recall from Section 8.4 that by executing AT generated from the complete test model using oracle O1, only one of 30 test suites (AT11) did not kill mutant M15. The remaining 29 test suites killed all of the 15 mutants. Switching the type of oracle to O2, two test suites (AT11 and AT12) did not kill mutant M15. Moreover, three mutants (M2, M12, and M14) were not killed by any test suites.

We also saw in Section 8.4 that the combination of ATP, the complete test model, and oracle O1, that all mutants were killed by every test suite. This was not the case, however, for oracle O2. Two mutants (M2, M14) were not killed by a single test suite. Three mutants (M7, M8, and M11) were killed by 29 test suites. Mutant M13 was killed by two test suites (ATP1 and ATP13). Finally, mutant M12 was killed by only one test suite (ATP14).

Also for test suites generated from the complete test model following the RTP coverage criterion in combination with oracle O1, results showed that all mutants were killed by each of the 30 test suites. By using oracle O2 on the other hand, the three mutants M2, M12, and M14 were not killed by any of the test suites.

As we now may have become aware of, LN2 did not provide high mutation scores. The cost on the other hand, was very low. Mutants M3–M11 and M15 were not killed by this combination of coverage criterion, test model, and oracle. Even worse results were obtained by applying oracle O2 to LN2; only mutants M1 and M13 were killed.

Bringing the test sequences one step deeper as compared to LN2, the combination of LN3, the complete test model and oracle O1 killed all mutants but M15. By applying oracle O2, the mutation score was reduced; M2, M12, and M14, in addition to M15 survived the tests.

Last, by using oracle O1 on tests generated from the complete test model following LN4 (i.e., four levels out from each edge of the initial state), no mutants survived. Again, by applying oracle O2, the mutation score was reduced; mutants M2, M12, and M14 were not killed.

To summarize, mutants M2, M12, and M14 were not killed by any of the six coverage criteria when using oracle O2. Let us look at these three mutants in particular. In mutant M2, the seeded fault regards an incorrect state invariant. This may explain why M2 was not killed by any coverage criteria when using the state-pointer oracle (i.e., O2). Regarding M12, the

seeded fault was erroneous on-entry behavior for a composite state, and the fault seeded in M14 was missing on-entry behavior. Checking only the state pointer will not detect such deviations from the specification.

9.5 Related Work

This section discusses the performance of the two oracles O1 and O2, and positions the results within existing work. Recall that O2 only evaluated the current state pointer as compared to O1 which also examined the expected state invariant (i.e., the abstract state).

As shown in Chapter 3, very few studies have compared oracles in the context of SBT; in fact, the study of Briand *et al.* [12] appears to be one of a kind in empirical comparison of oracles within this particular context. Although not being the only focus of their study, results revealed statistical significant differences between the two oracle strategies. Not surprisingly, they found that the precise oracle had stronger fault-detection ability than the state-invariant oracle. Moreover, explained by the fact that less attributes are checked, the state-invariant oracle required less resources during development and execution. Results varied from an improvement in fault detection from 11–72 percent for the original RTP strategy. Looking at the weaker form of RTP, cost was increased as little as just a few percent up to 300 percent.

Staats *et al.* [86, 88] explored how the selection of variables included in an oracle influences the effectiveness of the testing process. They proposed a method for supporting test oracle creation by automatically selecting oracle data (the set of variables monitored during testing). The purpose was to maximize the fault finding potential of the testing process with respect to the cost. Mutation analysis was used to rank variables in terms of fault-finding effectiveness. Experimental results from four commercial sub-systems from the civil avionics domain suggested that the approach may be cost-effective for producing small, effective oracle data. More precisely, they found improvements as high as 145.8 percent by using the new method as compared to monitoring all output variables.

In other fields, there are several studies that address this aspect, e.g., GUI testing [90, 91] where results revealed that employing expensive oracles leads to the detection of more faults using relatively few test cases. The importance of the test oracle's role in determining the cost and effectiveness of a testing strategy was demonstrated by Xie and Memon [91]. They found two interesting aspects of a test oracle in the context of GUI-based software – the expected output and the oracle procedure comparing expected outputs with actual outputs. They described a technique to specify different types of oracles by varying the level of detail in the

expected output and changing the oracle procedure. Results from an experiment comparing six oracles showed that the oracle has an important impact on the fault-detection ability. Weak oracles detect fewer faults. They also found that the best cost-benefit ratio is achieved by applying a stronger oracle at the end a test case execution. The use of thorough and frequently-executing test oracles can compensate for not having long test cases.

Table 28 Summary of related work

Reference	Objective	Research Method	Context	SUT	Oracle	Applied Testing strategy(ies)	No of mutants	Results
This study	To identify the influence of the test oracle on the cost-effectiveness of SBT by empirically evaluating two oracles.	Case study	Laboratory (Researcher)	Industrial	State invariant versus state pointer	AT, RTP, ATP, LN2, LN3, LN4	26	For all strategies, the state-invariant oracle (O1) obtained higher mutation score means than the state pointer (O2). Results show that when oracle O1 was used, significantly more time was spent on execution as compared to oracle O2. Significant differences were found in all data collected on mutation score; O1 always performed better than O2.
Briand *et al.* [12]	To empirically investigate the cost-effectiveness of SBT.	Three controlled experiments	Laboratory (Students)	Artificial	Precise versus state invariant	RTP (weak), RTP	(24, 25), 42, 81	Results suggest that the stronger precise oracle is more effective than the state-invariant oracle. Note however, that the increased fault-detection may come at a significant expense (test driver size).
Staats *et al.* [86, 88]	To support the creation of test oracles by automatically selecting oracle	Experiment	Laboratory (Researcher)	Industrial (4 industrial subsystems)	Which internal variables to be included in an oracle were randomly selected.	Branch coverage, MC/DC coverage. They randomly	200	Results demonstrate that the proposed method may be a cost-effective approach for producing small, effective oracle data, with fault finding improvements over current industrial

	data that improves fault-detection.				They also generated all oracles of size 1 (i.e., observing 1 variable).	generated 36 subsets of the randomized test suite, with subsets containing 10 to 1,000 tests from the original test suite.		best practice of up to 145.8% observed.
Xie *et al.* [91]	To investigate the importance of the oracle related to effectiveness and cost of the testing process.	Experiment	Laboratory (Researcher)	Artificial (4 software systems developed by students)	Six instances of test oracles were generated by varying the level of detail of oracle information (i.e., expected output) and oracle procedure (i.e., the procedure that compares oracle information with the actual output).	N/A 600 test cases (split in groups of 30)	100 faulty versions of each system	Results showed that the test oracles play an important role in determining the effectiveness and cost of the testing process. (1) Test cases significantly lose their fault detection ability when using "weak" test oracles; (2) in many cases, invoking a "thorough" oracle at the end of test case execution yields the best cost-benefit ratio; (3) certain test cases detect faults only if the oracle is invoked during a small "window of opportunity" during test execution; and (4) using thorough and frequently-executing test oracles can compensate for not having long test cases.

9.6 Discussion

As we now have seen, although little in quantity, existing research suggest that the applied oracle in testing has a large influence on the fault-detection ability [91, 86, 88, 12]. Table 28 summarizes the related work, which overall supports the findings of this study. For all strategies, the stronger oracle obtained significantly higher mutation scores than what were obtained by the weaker oracle.

Only the study of Briand *et al.* [12] is more precisely comparable as they also studied one of the coverage criteria in focus of this study, namely the RPT. Results for RTP indicated a 25 percent increase in fault-detection when using the strongest oracle. The study of Briand *et al.* differs from this study in the following matters: (1) The two oracles that were compared are not exactly the same. Both studies involve the state-invariant oracle. Nevertheless, Briand *et al.* compared the state-invariant oracle to a precise oracle; in this study, an oracle weaker than the state-invariant oracle was used as comparison. (2) Furthermore, in contrast to this study, the study of Briand *et al.* did not provide collected data on preparation time. (3) Briand *et al.* used students to perform the testing; in this study, the testing was carried out by the author. (4) The test suites were automatically generated in this study, but manually generated in the study of Briand *et al.* Finally, (5) the SUT was significantly larger in this study. Although Briand *et al.* [12] found great variations in the results, both studies suggest improvements in the fault-detection when using a stronger oracle – yet at a higher cost. For RTP, this regarded both preparation and execution time.

A common perception when it comes to oracles is that the better the oracle, the more expensive it is to use. However, there are no answers to the relation between preparation costs versus execution costs in terms of which parts of the process that takes more time when using a more complex oracle. In this case study, we saw that for both oracles, AT, RTP and LN4 performed best in terms of mutation score. Of those three strategies, RTP was the least expensive followed by LN4 and then AT. ATP was as effective as AT, RTP, and LN4, but only when applying O1; ATP in combination with O2 achieved a slightly lower mutation score. Note, however, that ATP was the most expensive strategy. Although a slightly lower mutation score was obtained for O1, LN3 also achieved a good cost-effectiveness – in particular when emphasizing cost. Finally, LN2 not only appeared to be the least expensive but also the weakest strategy from a fault-detection perspective. The most cost-effective

strategy in this case may seem to be RTP combined with oracle O1. Note, however, that LN3 combined with O1 obtained cost-effectiveness comparable to RTP.

9.7 Summary

In this chapter, we have seen results on how two different oracles influence the cost-effectiveness of state-based testing when using the strategies all transitions (AT), all round-trip paths (RTP), all transition pairs (ATP), paths of length 2 (LN2), paths of length 3 (LN3) and paths of length 4 (LN4) on a detailed state-based test model.

Descriptive statistics for the obtained results on cost and effectiveness was presented in Section 9.1. As changing the oracle will not have impact on the test-suite size, the test-suite size measure was excluded from this particular research question; sizes are determined by the testing strategies and the system specification, more precisely the number of transitions in the state-machine diagram. This leaves us with time as the surrogate measure of cost. Three of the six testing strategies spent slightly more time on preparing the test suites when combined with oracle O2 than with oracle O1. This regards LN3, RTP, and LN4. No difference between the oracles regarding preparation time was seen for LN2. Oracle O1 spent 7.5 percent more time than O2 for AT. Only a small relative difference (0.6 percent) was seen for ATP; O1 required more time on preparing the test suites as compared to O2.

For all strategies, oracle O1 obtained higher mutation score means than oracle O2. The greatest difference was found in results for LN2 (149.8 percent difference), followed by ATP (36.9 percent difference). Rather similar differences were seen among LN3 (27.2 percent), AT (25.3 percent), RTP (25 percent), and LN4 (25 percent).

Section 9.2 regarded statistical tests. The paired Wilcoxon signed-rank test was applied to the replicated data; that is, for AT, RTP, and ATP. Results show that when oracle O1 was used, significantly more time was spent on execution as compared to oracle O2. Significant differences were found in all data collected on mutation score; O1 always performed better than O2.

Overall, varying the oracle shows that there were significant differences in the preparation time for AT ($\alpha = 0.05$) and ATP ($\alpha = 0.01$); O1 spent more time than O2 on preparing the test suites. For RTP, on the other hand, no significant difference between the two oracles was found. Execution time was significantly higher when applying oracle O1. This pays off, however, in that O1 consistently achieves a higher mutation score.

For both oracles, AT, RTP and LN4 performed best in terms of mutation score. Of those three strategies, RTP was the least expensive followed by LN4 and then AT. ATP was as effective as AT, RTP, and LN4, but only when applying O1. ATP in combination with O2 achieved a slightly lower mutation score. Note, however, that ATP was the most expensive strategy. Although a slightly lower mutation score was obtained for O1, LN3 also achieved a good cost-effectiveness – in particular when looking at the cost. Finally, LN2 not only appeared to be the least expensive but also the weakest strategy from a fault-detection perspective.

As we have seen, the combinations of coverage criterion and oracle significantly impact the cost and fault-detection effectiveness of the testing strategies in different directions, in particular the execution cost. We found that the most cost-effective strategy in this study was RTP combined with oracle O1. Note, however, that LN3 combined with O1 obtained cost-effectiveness comparable to RTP.

Conclusions: Minor differences in preparation time were observed when applying oracle O2. Execution time, on the other hand, was significantly lower when applying oracle O2 for all six strategies. The large cost savings when using O2, however, had a negative impact on the effectiveness.

10 Case Study 3 – What is the Influence of the Test Model Abstraction Level on the Cost-Effectiveness?

In the two previous chapters, we have seen how six coverage criteria compares when applied to a detailed test model combined with two oracles of different strengths. The focus in this case study will be directed towards the test model itself with respect to the presence of details. This third case study seeks to answer how a less detailed test model as input to the test case generation will affect the cost-effectiveness:

RQ3: What is the influence of the test model abstraction level on the cost-effectiveness?

Results are collected by generating test suites and running tests on a less detailed test model as compared to the complete model used in Chapter 8 and Chapter 9, and organized similar to the structure of Chapter 9.

Recall that the main difference between the two test models is that the new test model, which will be investigated in this section, was abstracted one level up (i.e., the contents of every composite state were removed) – thereby the name, *the abstract model*. For more details about the test models, the reader is referred to Section 7.1.

Section 10.1 provides descriptive statistics regarding the impact of varying the oracle on cost and effectiveness, respectively. Statistical tests are presented in Section 10.2. Descriptive statistics from Chapter 8 and Chapter 9 are restated in this chapter for the purpose of making comparisons between the two test models and oracles. Section 10.3 presents an analysis of cost versus effectiveness. The survival of mutants is addressed in Section 10.4. Obtained results are further discussed towards existing research in Section 10.5. Furthermore, Section 10.6 discusses results from related work against results from this study. Finally, the obtained results are summarized in Section 10.7.

10.1 Descriptive Statistics

10.1.1 Descriptive Statistics for Cost

This section presents main features of the collected data on cost – the test-suite sizes and the time spent on preparing and executing these test suites for the abstract test model.

10.1.1.1 Test-Suite Size

Abstracting the test model one level, means that both sub states and the transitions between these sub states are removed from the state-machine diagram. As a consequence, the test-suite sizes are reduced.

Consider Table 29 which lists the test-suite sizes for the six testing strategies; both for the complete and the abstract test model, for oracles O1 and O2. Note that, in this chapter, test suite sizes are separated for O1 and O2 because we observed infeasible test cases in the abstract test suites. Removing the sub states made controlling the externally dependent variables harder. This led to infeasible test cases in the abstract test suites. The number of infeasible test cases observed for oracle O1 when applied with test suites generated from the abstract test model is higher than the suggested allowance (which is 10–15 percent for reduced coverage according to Binder [31]) – one exception is LN2 with 8 percent infeasible test cases. For oracle O2, the number of infeasible test cases was zero or less than 2 percent.

We see from Table 29 that no differences were observed between complete test suites when applied with O1 and O2. We also see that the abstract test model in general resulted in smaller test suites as compared to test suites generated from the complete test model. Figures vary from 14.8 percent to 89.8 percent smaller test suites for LN2 and AT, respectively, when generated from the complete model as compared to the abstract model when using O1. Using O2 instead resulted in a spread from 7.4 percent (LN2) to 80.1 percent (AT).

Table 29 Test-suite sizes - complete and abstract test model – oracles O1 and O2

Coverage Criterion	Test Model	Oracle	Test-Suite Size Total	Test-Suite Size Feasible	Infeasible Test Cases (%)	Diff O1 by O2 (%)	Diff Abstract by Complete O1 (%)	Diff Abstract by Complete O2 (%)
LN2	Complete	O1	27	27	0	0	-14.8	-7.4
		O2	27	27	0			
	Abstract	O1	25	23	8.0	-8.0		
		O2	25	25	0			
LN3	Complete	O1	143	143	0	0	-29.4	-14.0
		O2	143	143	0			
	Abstract	O1	123	101	17.9	-17.9		
		O2	123	123	0			
LN4	Complete	O1	764	764	0	0	-45.7	-23.6
		O2	764	764	0			
	Abstract	O1	585	415	29.1	-28.9		
		O2	585	584	0.2			
AT	Complete	O1	166	166	0	0	-89.8	-80.1
		O2	166	166	0			
	Abstract	O1	33	17	48.5	-48.5		
		O2	33	33	0			
RTP	Complete	O1	299	299	0.0	0	-77.9	-70.2
		O2	299	299	0.0			
	Abstract	O1	89	66	25.8	-25.8		
		O2	89	89	0			
ATP	Complete	O1	1425	1425	0	0	-86.5	-79.2
		O2	1425	1425	0			
	Abstract	O1	301	192	36.2	-35.1		
		O2	301	296	1.7			

10.1.1.2 Time

This section presents obtained timing results for the complete and abstract test model when combined with oracles O1 and O2. Table 30 and Table 31 show the descriptive statistics for preparation and execution time, respectively. Recall that time was measured in seconds.

Table 30 Descriptive statistics – preparation time

Coverage Criterion	Oracle	Model	Min (sec.)	Q1 (sec.)	Mean (sec.)	Median (sec.)	Q3 (sec.)	Max (sec.)	St Dev	N	Diff abstract by complete (%)	Diff abstract O2 by O1 (%)	Diff complete O2 by O1 (%)
AT	O1	abstract	173	180	222	192	210	972	143	30	-94.5	-3.8	-7.0
		complete	2,506	2,763	3,995	3,013	3,665	15,939	3,264	30			
	O2	abstract	168	175	213	185	202	965	143	30	-94.3		
		complete	2,249	2,474	3,715	2,766	3,394	15,663	3,280	30			
RTP	O1	abstract	182	184	204	193	210	309	30	30	-61.6	-4.7	0.2
		complete	484	512	531	525	545	607	28	30			
	O2	abstract	179	184	194	189	200	250	17	30	-63.5		
		complete	476	492	533	523	559	675	51	30			
ATP	O1	abstract	1,089	1,105	1,115	1,111	1,123	1,150	14	30	-96.1	2.3	-0.6
		complete	28,377	28,576	28,819	28,797	29,028	29,398	273	30			
	O2	abstract	1,088	1,097	1,141	1,116	1,161	1,368	64	30	-96.0		
		complete	28,305	28,409	28,641	28,504	28,787	29,548	345	30			
LN2	O1	abstract	83	83	83	83	83	83	-	1	-34.1	1.2	0.0
		complete	126	126	126	126	126	126	-	1			
	O2	abstract	84	84	84	84	84	84	-	1	-33.3		
		complete	126	126	126	126	126	126	-	1			
LN3	O1	abstract	315	315	315	315	315	315	-	1	-38.1	-1.9	1.4
		complete	509	509	509	509	509	509	-	1			
	O2	abstract	309	309	309	309	309	309	-	1	-40.1		
		complete	516	516	516	516	516	516	-	1			
LN4	O1	abstract	3,943	3,943	3,943	3,943	3,943	3,943	-	1	-25.5	1.3	3.8
		complete	5,295	5,295	5,295	5,295	5,295	5,295	-	1			
	O2	abstract	3,993	3,993	3,993	3,993	3,993	3,993	-	1	-27.3		
		complete	5,494	5,494	5,494	5,494	5,494	5,494	-	1			

Table 31 Descriptive Statistics – Execution Time

Coverage Criterion	Oracle	Model	Min (sec.)	Q1 (sec.)	Mean (sec.)	Median (sec.)	Q3 (sec.)	Max (sec.)	St Dev	N	Diff abstract by complete (%)	Diff abstract O2 by O1 (%)	Diff complete O2 by O1 (%)
AT	O1	abstract	29	42	64	57	86	129	27	30	-97.4	-73.8	-83.1
		complete	1,765	2,108	2,455	2,377	2,785	3,617	469	30			
	O2	abstract	9	12	17	16	19	37	7	30	-95.9		
		complete	293	358	415	417	473	571	71	30			
RTP	O1	abstract	52	62	72	74	76	100	11	30	-85.2	-69.4	-80.5
		complete	341	460	489	504	524	607	54	30			
	O2	abstract	14	19	22	21	23	47	7	30	-76.9		
		complete	80	88	95	97	101	119	9	30			
ATP	O1	abstract	215	342	395	398	448	528	75	30	-88.2	-70.6	-83.0
		complete	2,607	2,972	3,341	3,381	3,669	3,978	429	30			
	O2	abstract	65	100	116	117	127	177	22	30	-79.6		
		complete	429	493	569	547	654	773	101	30			
LN2	O1	abstract	16	16	16	16	16	16	-	1	-11.1	-68.8	-72.2
		complete	18	18	18	18	18	18	-	1			
	O2	abstract	5	5	5	5	5	5	-	1	0.0		
		complete	5	5	5	5	5	5	-	1			
LN3	O1	abstract	85	85	85	85	85	85	-	1	-37.5	-68.2	-79.4
		complete	136	136	136	136	136	136	-	1			
	O2	abstract	27	27	27	27	27	27	-	1	-3.6		
		complete	28	28	28	28	28	28	-	1			
LN4	O1	abstract	667	667	667	667	667	667	-	1	-21.5	-72.6	-79.6
		complete	850	850	850	850	850	850	-	1			
	O2	abstract	183	183	183	183	183	183	-	1	5.8		
		complete	173	173	173	173	173	173	-	1			

Starting with the preparation time, observations show that it takes significantly shorter time for all strategies to prepare test suites from the abstract test model as compared to the complete test model. The greatest difference was seen for ATP (96 percent for both O1 and O2), closely followed by AT (95 percent for O1 and 94 percent for O2). RTP was reduced by 62 percent for O1 and 64 percent for O2. Also LN2 and LN3 were prepared in less time from the abstract model – LN2: 34 percent for O1 and 33 percent for O2; LN3: 38 percent for O1 and 40 percent for O2. Slightly less difference was seen in the results for LN4: 26 percent and 27 percent for O1 and O2.

Studying the execution time in Table 31, we see a similar trend with great differences between the times spent on executing the test suites generated from the abstract model as compared to the complete model. The largest reduction was achieved by AT (97 percent for O1 and 96 percent for O2), followed by ATP (88 percent for O1 and 80 percent for O2), and RTP (85 percent for O1 and 77 percent for O2). When O1 was combined with the abstract model for LN2, LN3, and LN4, results show that the preparation times were reduced by 11 percent, 38 percent, and 22 percent, respectively. No difference was found between the abstract and complete model when applying O2 to LN2. A minor reduction was found for LN3 (4 percent). The execution time for LN4, on the other hand, was increased by 6 percent.

Regarding LN2, LN3, and LN4, we see an overall smaller difference between the two models. This, however, can be explained by the first surrogate measure of cost, test-suite size. Recall from Table 29 that significantly smaller test-suite sizes were found for the abstract model versus the complete model for LN2, LN3, and LN4 contra AT, RTP, and ATP. Thus, lower differences in execution and preparation time would be expected results.

Comparing the two oracles applied to the abstract test model provided rather consistent results to what was found when applying the two oracles to the complete test model. Differences in preparation time for the abstract model was in the range from -5 percent to 2 percent compared to -7 percent to 4 percent for the complete model. Regarding execution time, we see a similar correlation between the abstract and the complete model: for the abstract model, test suites combined with O2 spent from 68 percent to 74 percent less time on execution compared to O1, whereas test suites generated from the complete model combined with O2 spent 72 percent to 83 percent less time on execution compared to O1.

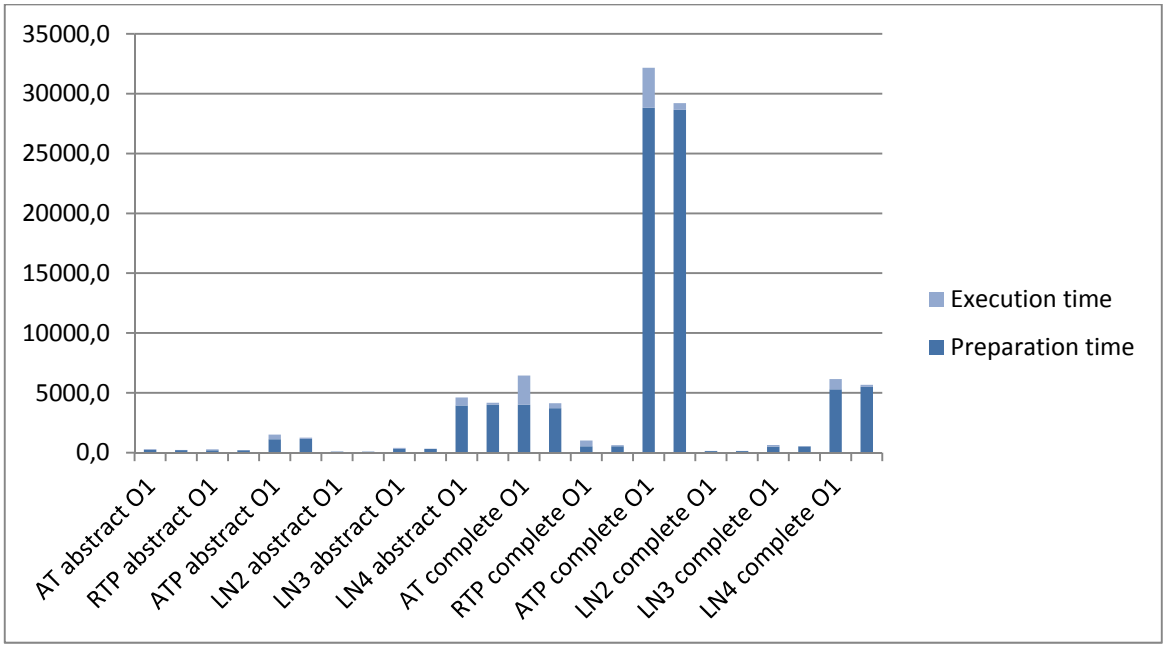


Figure 37 Preparation and execution time in seconds

Figure 37 displays a graphical representation of the means. The purpose of this figure is only to give a perspective of the relationship between the positioning of each strategy and oracle. Numbers can be found in Appendix G (complete model, oracle O1), Appendix H (complete model, oracle O2), Appendix I (abstract model, oracle O1), and Appendix J (abstract model, oracle O2). The figure shows a significant increase in preparation time for ATP combined with the complete test model and oracles O1 and O2. Other peaks are found for the four LN4 combinations, and for AT in combination with the complete test model and oracles O1 and O2.

The Means of AT, RTP, and ATP

We will now investigate the distributions of the observations used to generate the means for AT, RTP, and ATP. Figure 38 presents the performance of AT, RTP, and ATP regarding preparation time for oracle O1 when replicated 30 times using different test trees.

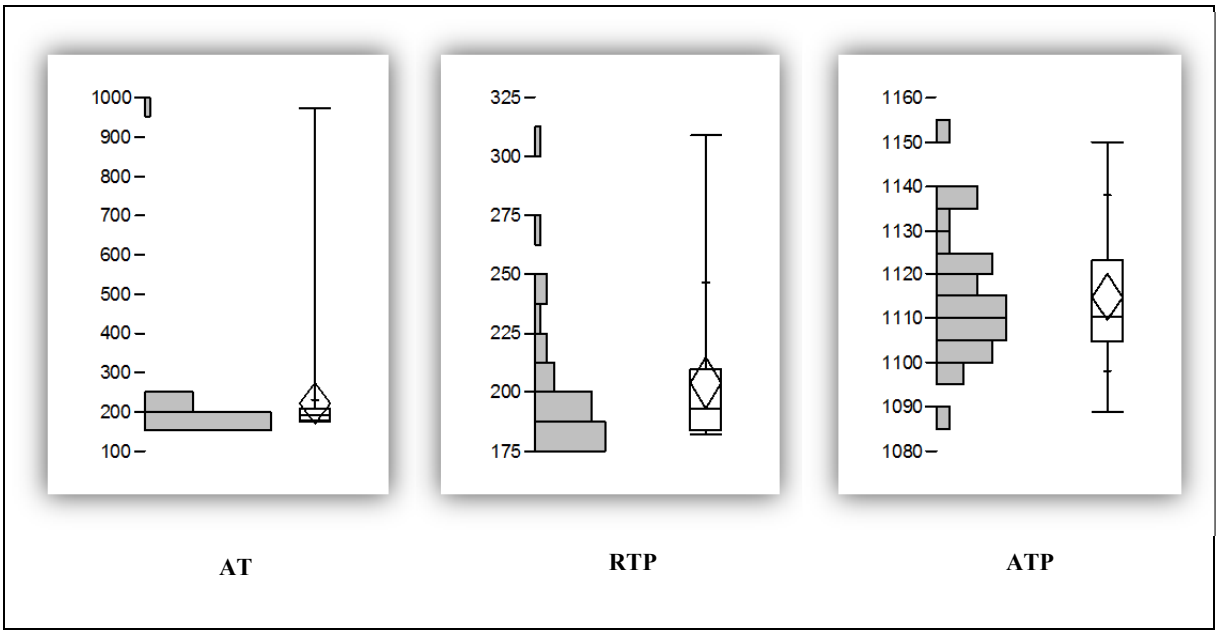


Figure 38 Distribution – preparation time – abstract O1

Let us start with the distribution of AT. The histogram and box plot show that 29/30 observations were found between 173 to 246 seconds. Being the main reason for the presence of a high standard deviation, one outlier was found at 972 seconds.

The distribution of RTP is skewed to the left. Two outliers were observed at 271 and 309 seconds. The minimum time spent on preparation was 182 seconds, the mean was 204 seconds, whereas the maximum value was found at 309 seconds.

Closer to the shape of a normal distribution, ATP is spread from 1,089 seconds to 1,150 seconds, having its mean at 1,115 seconds.

The means and medians of the AT and RTP distributions were rather closely located – the means were slightly higher than the medians due to the few outliers in the upper range. The median of AT was 13.5 percent lower than its mean; the median of RTP was 5.4 percent lower than its mean. ATP, on the other hand only differed with 0.36 percent.

Table 32 Inter quartile ranges – preparation time – abstract O1

Coverage Criterion	Q1	Q3	Q3-Q1	IQR	Lower limit	Upper limit
AT	180	210	31	46	133	257
RTP	184	210	26	39	145	249
ATP	1,105	1,123	19	28	1,077	1,151

Applying oracle O2 results in the distributions displayed in Figure 39. The AT distribution has a quite similar shape to AT when applied with oracle O1 (Figure 38). Preparing the AT

test suite from the abstract model using O2 only reduced the time with 3.8 percent. The minimum time was 168 seconds, the mean 213 seconds, and the maximum time 965 seconds.

For RTP, the preparation time was reduced by 4.7 percent when applying oracle O2. The shapes of the distributions for oracles O1 and O2 are rather similar, but of course slightly lower minimum and maximum values for O2.

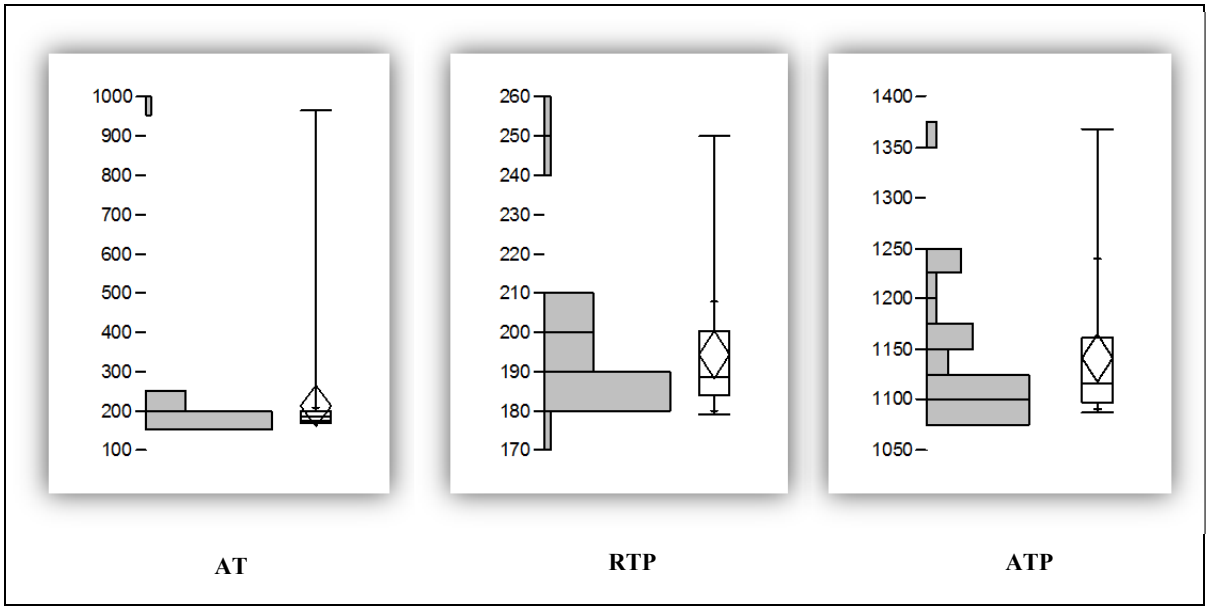


Figure 39 Distribution – preparation time – abstract O2

Apparently, the distribution of ATP when applying oracle O2 has to some extent a different shape than what was seen for oracle O1. Yet, both distributions are centered very close, around 1,141 seconds for O2 versus 1,115 for O1. In the O2 distribution, however, we see, according to Figure 39 and Table 33, an outlier at 1,368 seconds. Sixteen out of thirty observations were within the lower and upper quartiles for oracle O1. For O2 on the other hand, as much as 22/30 observations were within the lower and upper quartiles.

Table 33 Inter quartile ranges – preparation time – abstract O2

Coverage Criterion	Q1	Q3	Q3-Q1	IQR	Lower limit	Upper limit
AT	175	202	27	40	135	242
RTP	184	200	16	24	160	225
ATP	1097	1161	65	97	1000	1258

Figure 40 displays the spread in the collected execution-time data from running test suites generated from the abstract model using oracle O1.

The execution times for AT varies from the minimum observation 29 seconds to the maximum observation 129 seconds. Being the reason for an 11.5 percent higher mean (64.4 seconds) than median (57 seconds), the seven observations above Q3 had a much higher spread than the remaining observations below Q1. That is, we see a distribution skewed to the left, having 16/30 observations within Q1 and Q3.

The minimum value in the RTP distribution was 52 seconds, whereas the mean was 72.1 seconds. According to Table 34, there was one outlier in the RTP observations at 100 seconds, which also was the maximum observed value. In this distribution, the median was slightly higher than the mean due to more observations in the lower end of the possible value range.

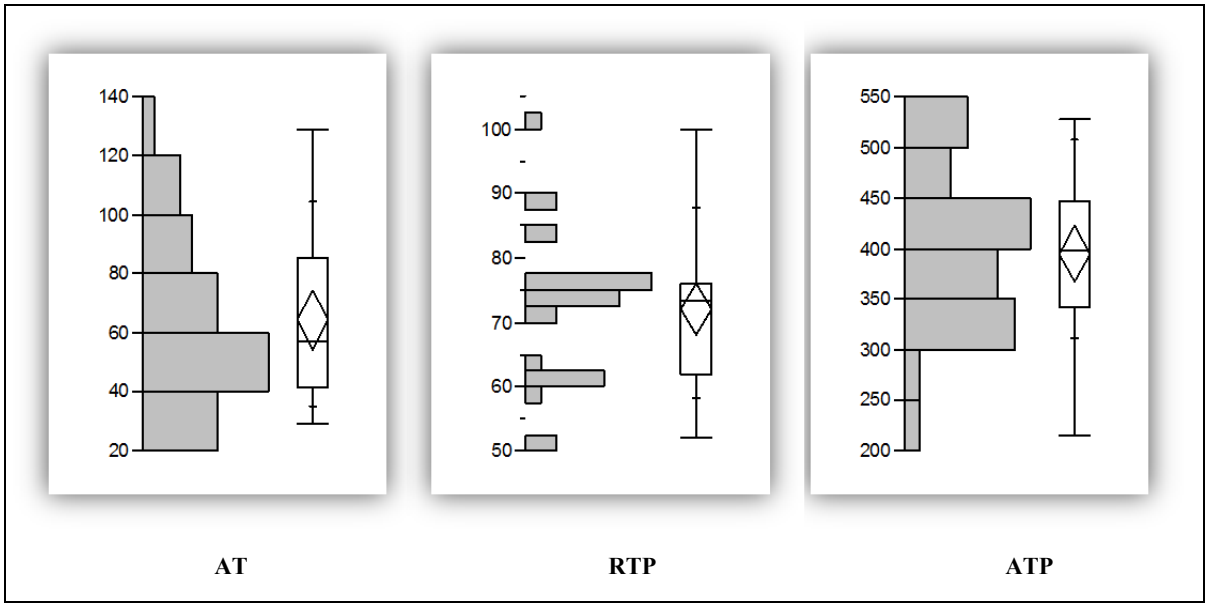


Figure 40 Distribution – execution time – abstract O1

ATP's distribution was spread from 215 seconds to 528 seconds, having its mean positioned at 394.9. The median is only 0.65 percent lower. No outliers were found in the distributions of AT and ATP.

Table 34 Inter quartile ranges – execution time – abstract O1

Coverage Criterion	Q1	Q3	Q3-Q1	IQR	Lower limit	Upper limit
AT	42	86	44	66	-25	152
RTP	62	76	14	21	41	97
ATP	342	448	106	159	183	606

The final set of distributions regard test suites generated from the abstract model when using oracle O2. Starting with AT, Figure 41 shows a distribution slightly skewed to the left. The minimum value was observed at 9 seconds, whereas the maximum time was 37 seconds. The median (16.9 seconds) was 8.3 percent lower than the mean (15.5 seconds). Two outliers were found in the upper end of the collected data at 36 and 37 seconds.

Continuing with RTP, we see a distribution where the minimum observed time to execute the test suite at 14 seconds and the maximum time at 47 seconds. The sample presents a median (21 seconds) that was 5 percent lower than the mean (22.1 seconds). No outliers are indicated by Table 35.

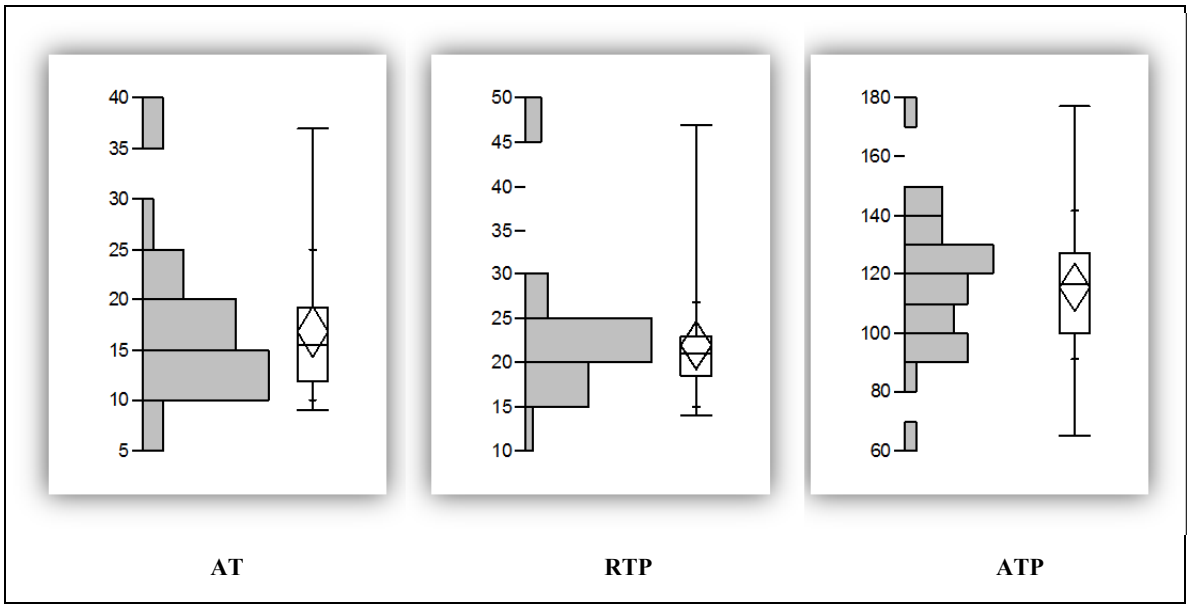


Figure 41 Distributions – execution time – abstract O2

Last but not least, the distribution of ATP. Observations are positioned in a range between 65 seconds and 177 seconds. The mean (115.9 seconds) was only 0.94 percent lower than the median (117 seconds). Again, by looking at Table 35, one outlier is suggested at 177 seconds.

Table 35 Inter quartile ranges – execution time – abstract O2

Coverage Criterion	Q1	Q3	Q3-Q1	IQR	Lower limit	Upper limit
AT	12	19	7	11	1	30
RTP	19	23	5	7	12	30
ATP	100	127	27	41	59	168

10.1.2 Descriptive Statistics for Effectiveness

Table 36 presents the main features of the collected data on mutation score.

Table 36 Descriptive statistics for mutation score

Coverage Criterion	Oracle	Model	Min	Q1	Mean	Median	Q3	Max	St Dev	N	Diff abstract by complete (%)	Diff abstract O2 by O1 (%)	Diff complete O2 by O1 (%)
AT	O1	abstract	0.000	0.200	0.262	0.200	0.200	0.800	0.168	30	-73.7	-71.3	-20.2
		complete	0.900	1.000	0.997	1.000	1.000	1.000	0.018	30			
	O2	abstract	0.000	0.000	0.075	0.000	0.050	0.530	0.151	30	-90.5		
		complete	0.730	0.800	0.795	0.800	0.800	0.800	0.018	30			
RTP	O1	abstract	0.870	0.870	0.870	0.870	0.870	0.870	0.000	30	-13.0	-31.0	-20.0
		complete	1.000	1.000	1.000	1.000	1.000	1.000	0.000	30			
	O2	abstract	0.600	0.600	0.600	0.600	0.600	0.600	0.000	30	-25.0		
		complete	0.800	0.800	0.800	0.800	0.800	0.800	0.000	30			
ATP	O1	abstract	0.867	0.867	0.867	0.867	0.867	0.867	0.000	30	-13.3	-27.2	-26.9
		complete	1.000	1.000	1.000	1.000	1.000	1.000	0.000	30			
	O2	abstract	0.600	0.600	0.631	0.600	0.667	0.667	0.034	30	-13.6		
		complete	0.600	0.730	0.731	0.730	0.730	0.800	0.035	30			
LN2	O1	abstract	0.267	0.267	0.267	0.267	0.267	0.267	-	1	-20.0	-100.0	-60.0
		complete	0.333	0.333	0.333	0.333	0.333	0.333	-	1			
	O2	abstract	0.000	0.000	0.000	0.000	0.000	0.000	-	1	-100.0		
		complete	0.133	0.133	0.133	0.133	0.133	0.133	-	1			
LN3	O1	abstract	0.867	0.867	0.867	0.867	0.867	0.867	-	1	-7.1	-30.8	-21.4
		complete	0.933	0.933	0.933	0.933	0.933	0.933	-	1			
	O2	abstract	0.600	0.600	0.600	0.600	0.600	0.600	-	1	-18.2		
		complete	0.733	0.733	0.733	0.733	0.733	0.733	-	1			
LN4	O1	abstract	0.867	0.867	0.867	0.867	0.867	0.867	-	1	-13.3	-30.8	-20.0
		complete	1.000	1.000	1.000	1.000	1.000	1.000	-	1			
	O2	abstract	0.600	0.600	0.600	0.600	0.600	0.600	-	1	-25.0		
		complete	0.800	0.800	0.800	0.800	0.800	0.800	-	1			

Table 36 displays the results for each testing strategy, both test models and oracles. First of all, let us consider the abstract versus the complete models. An overall trend in the results is that the abstract test models obtained lower mutation scores (mean values) than what were achieved by the complete test models.

In more detail, the largest differences in the ability of killing mutants between the abstract and complete test models were observed in results for LN2 and AT. By looking at the results for LN2 combined with oracle O2, we see that no mutants were killed. In comparison, the complete model provided test cases that killed 2/15 mutants. The mutation score mean for AT combined with O2 was 0.076 as compared to 0.795 for the complete test model. This means that the abstract model performed 90.5 percent worse than the complete model. Combining the AT strategy with the abstract model and oracle O1 provides a slight improvement, although a 73.7 percent reduction was found in mutation score (0.262 for the abstract model as compared to 0.997 for the complete model). For both RTP and LN4 combined with oracle O2, the mutation score was 25 percent reduced as compared to the complete model. Test suites generated from the abstract model for LN2 with O1 and LN3 with O2 performed 20 percent and 18.2 percent weaker than the test suites generated from the complete models. ATP O2 (13.6 percent), ATP O1 (13.3 percent), LN4 O1 (13.3 percent), and RTP O1 (13 percent) provided quite similar results. Even less difference was found between the abstract and complete model for strategy LN3 when combined with oracle O1; the abstract model only performed 7.1 percent worse than the complete model.

Focusing on the test suites generated from the abstract models only, we see that oracle O1 performed significantly better than oracle O2. This regards all six strategies. The best improvement with respect to mutation score was found for LN2; no mutants were killed by oracle O2 whereas 4/15 mutants were killed by oracle O1. On average, 1.13 mutants were killed by O2 in combination with AT – in comparison, almost four mutants (3.94) were killed by oracle O1 on average. For RTP, LN3, and LN4, we see that oracle O2 obtained around 31 percent lower mutation scores than oracle O1. Regarding ATP we see a 27.2 percent reduction in mutation score. The mutation-score means for all combinations, as displayed in Table 36, are graphically visualized in Figure 42.

.

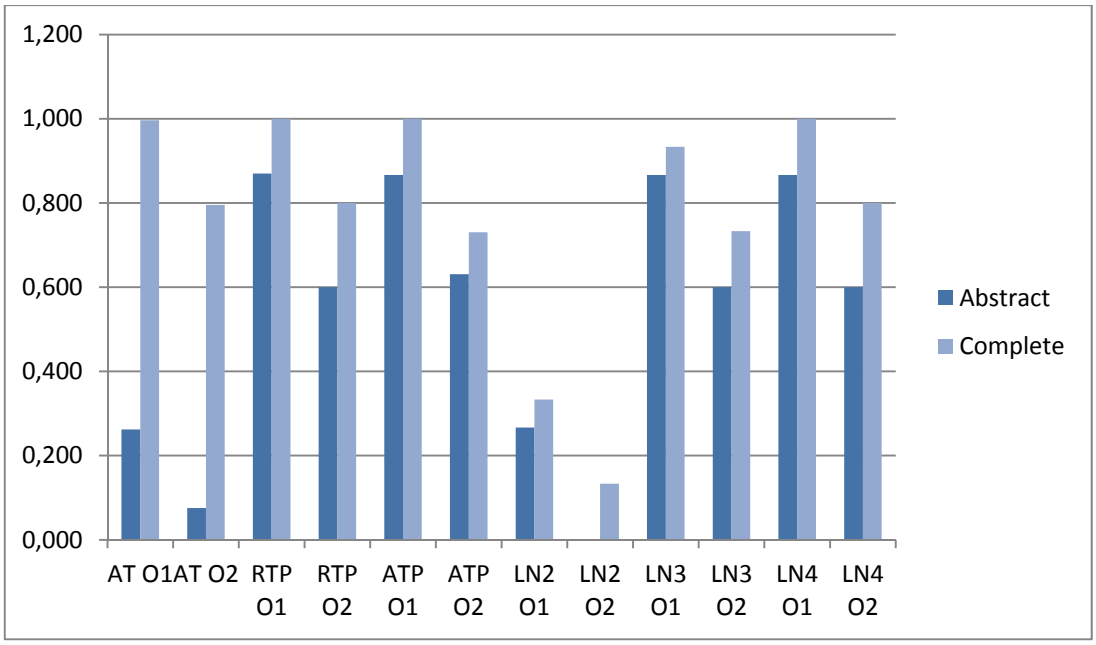


Figure 42 Mutation score means

Comparing Means

This section investigates the replications of AT, RTP, and ATP for the purpose of identifying possible statistical variability in the collected data on mutation scores. Figure 43 displays the histograms and box plots.

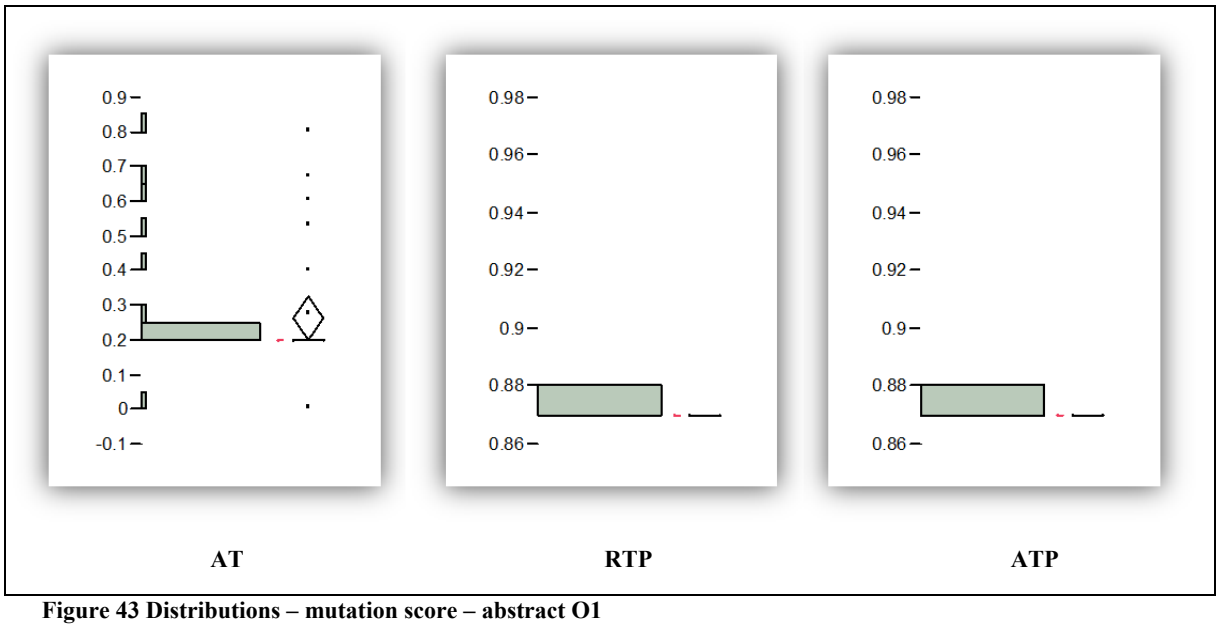


Figure 43 Distributions – mutation score – abstract O1

The main part of the AT observations (76.7 percent) had 0.2 as their mutation scores. The remaining test suites achieved mutation scores spread from 0 to 0.8. Only one test suite did

not detect any faults. According to Table 37, observations below and above 0.2 are all outliers.

Both RTP (0.87) and ATP (0.87) obtained the exact same mutation score for every test suite. The reason for this is that the trees from which the test suites are generated from, provide the same sequences, but in different orders. Thus, the result will always be the same. It is, however, also possible to generate truly different trees – i.e., trees with actually different sequences, not only the order of the sequences.

Table 37 Inter quartile ranges – mutation score – abstract O1

Coverage Criterion	Q1	Q3	Q3-Q1	IQR	Lower limit	Upper limit
AT	0.20	0.20	0.00	0.00	0.20	0.20
RTP	0.87	0.87	0.00	0.00	0.87	0.87
ATP	0.87	0.87	0.00	0.00	0.87	0.87

Figure 44 presents the distributions of mutation scores achieved by test suites generated from the abstract model applying oracle O2.

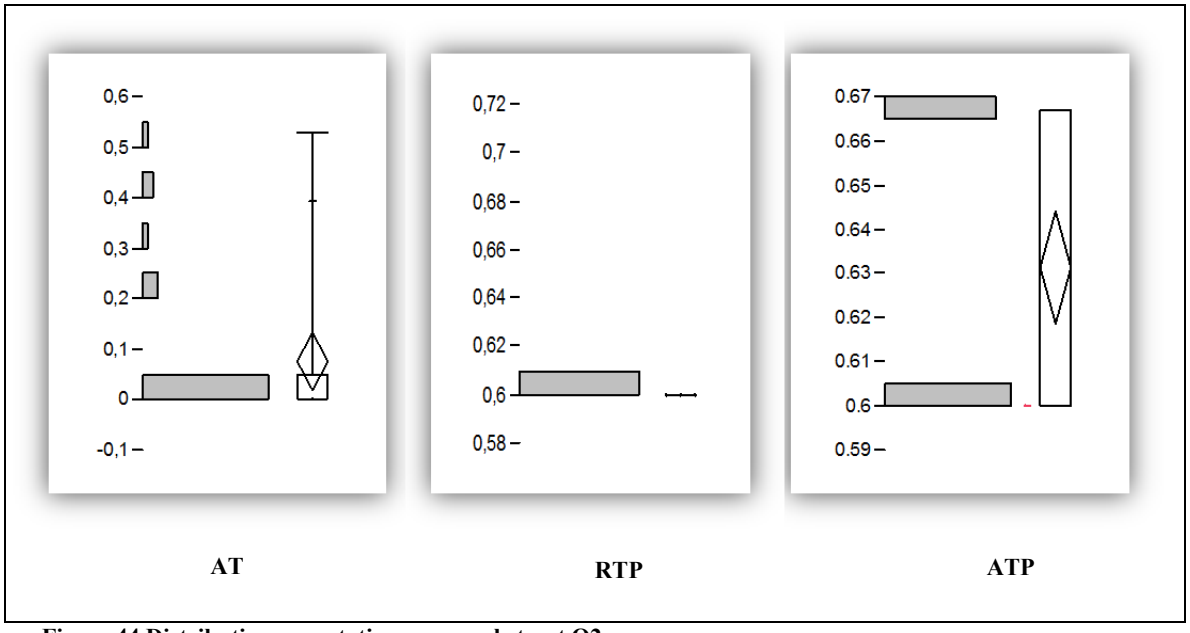


Figure 44 Distributions – mutation score – abstract O2

From the histogram of AT's distribution, we see that twenty-four test suites did not kill any mutants. On average, three test suites killed three mutants, one killed 4.95 mutants, two killed six mutants, and one killed 7.95 mutants. The IQR displayed in Table 38 shows that there are seven outliers. The distribution of AT combined with O2 has a similar shape to what was seen for AT combined with O1. However, the obtained results were lower on average

Similar to RTP applied with O1, yet lower, the 30 generated RTP test suites again provided the same mutation score (0.6) for all test suites.

The final distribution, ATP, a mean mutation score of 0.63 was achieved. The results among the test suites were spread from 0.6 to 0.67. There were more variations in the collected data on mutation score for ATP applied with oracle O2 than with oracle O1, in addition to the mean being lower for O2.

Table 38 Inter quartile ranges – mutation score – abstract O2

Coverage Criterion	Q1	Q3	Q3-Q1	IQR	Lower limit	Upper limit
AT	0.000	0.050	0.050	0.075	-0.075	0.125
RTP	0.600	0.600	0.000	0.000	0.600	0.600
ATP	0.600	0.667	0.067	0.101	0.500	0.768

10.2 Statistical Tests

This section provides statistical tests on the data described by descriptive statistics in Section 10.1. The purpose of the tests is to confirm/reject the hypothesis:

H_0: There are no significant differences in cost and effectiveness for the strategies AT, RTP, and ATP when varying the level of details in the test models.

Tests are run in R [127]. The actual tests are listed in Appendix M.

Due to the non-existing relation between test paths for each of the 30 replications in the complete versus the abstract model, these cannot be considered as pairs in statistical tests. Hence, the non-paired Wilcoxon signed-rank test was used for confirming/rejecting statistical significant difference.

Table 39 displays results from comparisons of collected data on preparation time for abstract versus complete model. We see that the complete model, both for O1 and O2, required more preparation time than for the abstract model. The very low *p*-values (< 2.2e-16 for both O1 and O2) show that there is a significant difference. Moreover, the measure of stochastic superiority tells us that the probability that preparation times for the abstract model is higher than the complete model is 11.5 percent, regardless of applied oracle. This means that the effect size is large.

Table 39 Non-paired Wilcoxon signed-rank test comparing abstract and complete model – preparation time

H_0	Coverage Criterion	Oracle	Measure	*p*-value	$\hat{A}_{12}$	Effect Size	Result	Sign. Diff. (CI)
Abstract = Complete	AT, RTP, ATP	O1	Prep. time	< 2.2e-16	0.115	Large	Abstract < Complete	Yes (99 %)
Abstract = Complete	AT, RTP, ATP	O2	Prep. time	< 2.2e-16	0.115	Large	Abstract < Complete	Yes (99 %)

Similar results were found when running tests on execution time – test suites generated from the complete model spent more time on execution than from the abstract model. Again, differences were seen at a 0.01 significance level (p< 2.2e-16). Furthermore, the effect size was even larger as compared to the preparation time; for O1 we see that the probability of having higher execution times for the abstract model than the complete model is only 1.7 percent. Applying oracle O2, the probability was 9.1 percent.

Table 40 Non-paired Wilcoxon signed-rank test comparing abstract and complete model – execution time

H_0	Coverage Criterion	Oracle	Measure	*p*-value	$\hat{A}_{12}$	Effect Size	Result	Sign. Diff. (CI)
Abstract = Complete	AT, RTP, ATP	O1	Exec. time	< 2.2e-16	0.017	Large	Abstract < Complete	Yes (99 %)
abstract = complete	AT, RTP, ATP	O2	Exec. time	< 2.2e-16	0.091	Large	Abstract < Complete	Yes (99 %)

Table 41 presents obtained results for mutation score. Significant differences were found both when applying oracle O1 and O2 at a 0.01 significant level. The effect size was large in both cases, which means that there is higher probability of achieving a better mutation score by using a complete model when generating test suites.

Table 41 Non-paired Wilcoxon signed-rank test comparing abstract and complete model – mutation score

H_0	Coverage Criterion	Oracle	Measure	p-value	$\hat{A}_{12}$	Effect Size	Result	Sign. Diff. (CI)
abstract = complete	AT, RTP, ATP	O1	Mut. score	< 2.2e-16	0	Large	Abstract < Complete	Yes (99 %)
abstract = complete	AT, RTP, ATP	O2	Mut. score	< 2.2e-16	0.005	Large	Abstract < Complete	Yes (99 %)

Overall, i.e., considering the three strategies combined with each of O1 and O2, Table 42 shows that preparation time, execution time and mutation score are significantly different for test suites generated from the abstract compared to test suites generated from the complete model. Significance was found for $\alpha = 0.01$. As we can see, the effect size is large for preparation time, execution time, and mutation score.

Table 42 Non-paired Wilcoxon signed-rank test comparing abstract and complete model – all measures

H_0	Coverage Criterion	Oracle	Measure	p-value	$\hat{A}_{12}$	Effect Size	95 % CI for $\hat{A}_{12}$	99 % CI for $\hat{A}_{12}$	Result	Sign. Diff. (CI)
abstract = complete	AT, RTP, ATP	O1, O2	Prep. time	< 2.2e-16	0.115	Large	[0.086, 0.152]	[0.078, 0.166]	Abstract < Complete	Yes (99 %)
abstract = complete	AT, RTP, ATP	O1, O2	Exec. time	< 2.2e-16	0.075	Large	[0.054, 0.104]	[0.048, 0.114]	Abstract < Complete	Yes (99 %)
abstract = complete	AT, RTP, ATP	O1, O2	Mut. score	1.027e-10	0.170	Large	[0.132, 0.217]	[0.121, 0.233]	Abstract < Complete	Yes (99 %)

The null hypothesis was thus rejected by the statistical tests, and we can conclude that there are significant differences in the cost-effectiveness for the strategies AT, RTP, and ATP when varying the level of details in the test model.

10.3 Cost-Effectiveness

This section presents how raising the test model abstraction level affects the cost-effectiveness. Figure 45 positions the cost[9]-effectiveness for the 24 combinations of coverage criteria, oracles, and test models. Each combination of oracle and test model is represented by a unique color: sky blue represents oracle O1 and the complete test model; pale blue represent oracle O2 and the complete test model; oakes yellow represents oracle O1 and the abstract test model; and finally, salmon pink represents oracle O2 and the abstract test model.

Take a look at the lowest part of the figure at the left hand side. Not surprisingly, we observe for LN2 that by reducing the level of details in the test model, both for oracle O1 and O2, the mutation score is lowered even more. Differences of great impact on both cost and effectiveness were seen for AT when reducing the abstraction level. None of the combinations of LN2 can be recommended; neither the combinations of AT and the abstract test model.

The influence of the abstract test model on ATP shows a major reduction in cost, yet retaining an overall high mutation score when using oracle O1. Looking at the results for LN4, we see that there are significant differences in the cost, both for test-suite size and time. The reduction in mutation score is also present for both combinations of test model and oracle (i.e., complete test model and oracle O1 versus abstract test model and oracle O2). Interestingly, however, we see that the abstract model in combination with the better oracle (O1) actually performs better with respect to fault-detection as compared to the complete model combined with oracle O2. The latter also regards LN2, LN3, RTP, and ATP.

Results for RTP when generating test suites from the complete model using oracle O1 show similar fault-detection as compared to LN3 (complete model, oracle O1), AT (complete model, oracle O1), LN4 (complete model, oracle O1), and ATP (complete model combined with oracle O1). The cost, however, differ. Notice that the cost of RTP is lower than for AT LN4, and ATP (complete model).

[9] Please note that time is shown as the total of preparation time and execution time

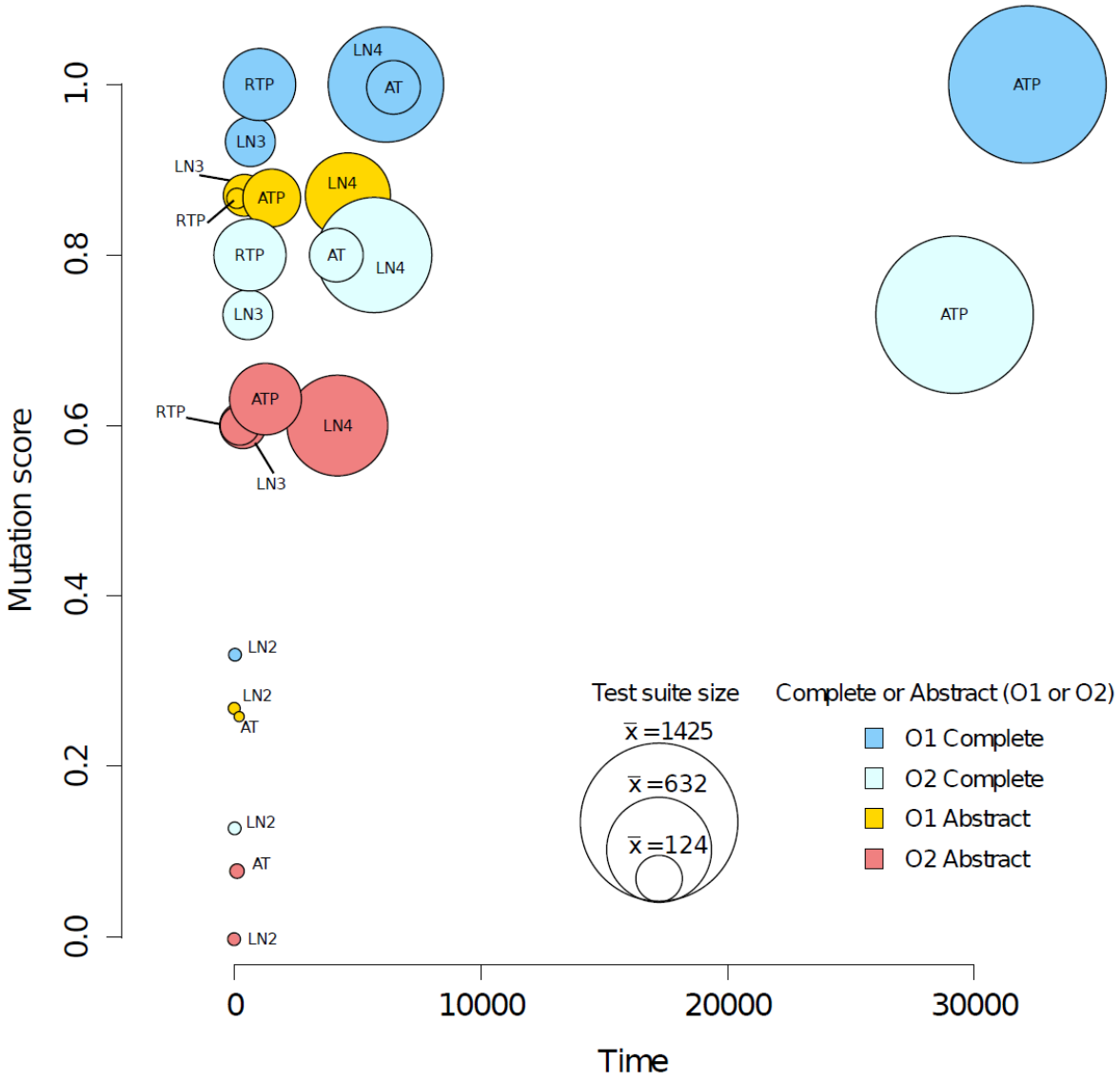


Figure 45 Abstract versus complete test model – mutation score, test-suite size and time

For each combination of coverage criterion, test model, and oracle, Figure 46– Figure 49 show the number of killed mutants divided by each of the surrogate measures on cost.

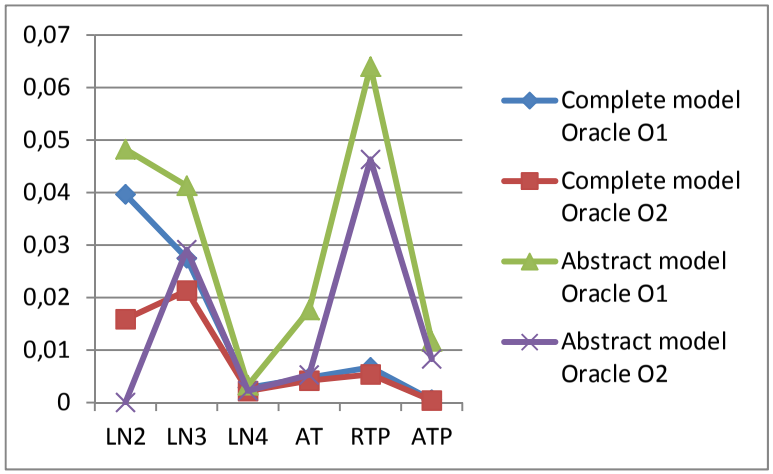


Figure 46 Number of mutants killed over preparation time

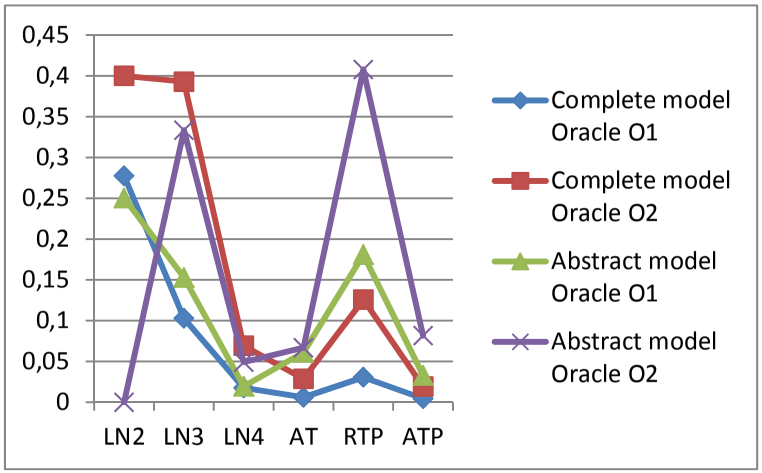


Figure 47 Number of mutants killed over execution time

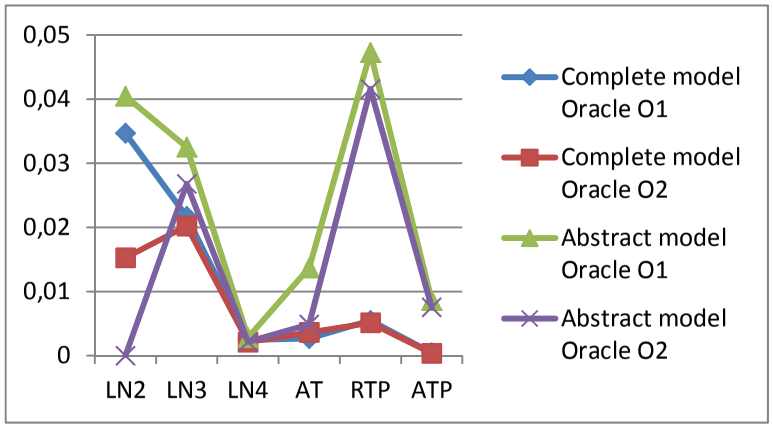


Figure 48 Number of mutants killed over total time

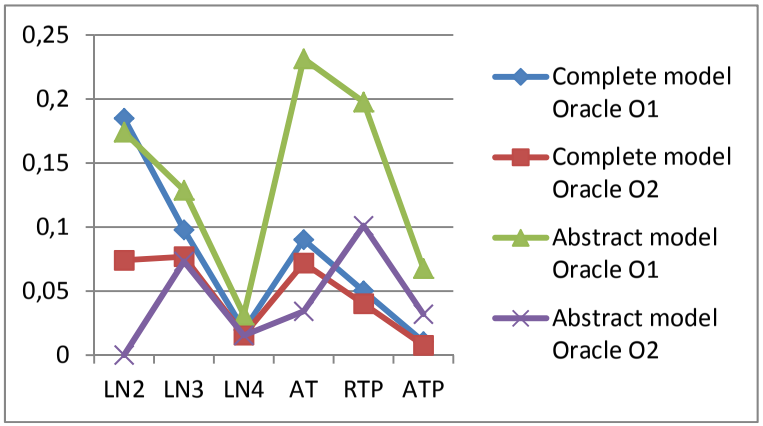


Figure 49 Number of mutants killed over test-suite sizes

10.4 Analysis of Mutant Survival

We will now take a closer look at the mutants not killed by the remaining testing strategy combinations that have not already been discussed in Chapter 8 or 9; that is the test suites generated from the abstract test model.

Only two mutants (M12 and M15) were not killed by any of the AT test suites in combination with the O1 oracle. Mutants M1, M2, and M14 were killed by 29/30 test suites. Only one test suite (AT9) killed M9. The other mutants were each killed by 3–4 test suites. Killing 12 mutants, test suite AT11 achieved the highest mutation score among the AT test suites. Only M9, M12, and M15 were not killed by AT11.

Six mutants (M1, M2, and M12–M15) were not killed by any of the AT test suites when using oracle O2. Eight mutants (M3–M8, M10, and M11) were killed by four test suites. Mutant M9 was killed by two test suites (AT29 and AT9).

Two mutants were not killed by any of the ATP test suites when using oracle O1, namely M12 and M15. Combining ATP with O2, on the other hand, resulted in more mutants remaining un-killed: M1, M2, and M12–M15.

M12 and M15 were not killed by any of the RTP test suites when executed together with the O1 oracle. The remaining mutants were all killed by each of the 30 test suites. No RTP test suite in combination with oracle O2 could kill any of the six mutants M1, M2, and M12–M15.

Eleven mutants (M3–M12, and M15) were not killed by the LN2 test suites when used with oracle O1. By using oracle O2, none of the 15 mutants were killed.

Neither the LN3 test suites in combination with oracle O1 killed M12 and M15. Using oracle O2, even more mutants remained un-killed: in addition to M12 and M15, also M1, M2, M13, and M14 were not detected.

Analyzing the LN4 test suites used with oracle O1 showed that two mutants M12 and M15 were not killed. Combined with oracle O2, the following mutants could not be killed: M1, M2, and M12–M15.

To summarize, results show that certain mutants were harder to kill by the test suites generated from the abstract test model. Regardless of coverage criterion applied and oracle used, M12 and M15 were not killed by any of the test cases generated from the abstract test model. As for oracle O2, in addition to M12 and M15, mutants M1, M2, M13, and M14 were not killed by any of the test suites. The seeded faults in these mutants all concerned sub states

that were removed from the test model. This explains why the test cases generated from the abstract test model could not kill those mutants.

10.5 Related Work

Reducing the test-suite size by abstracting the test model is yet another area of related work where few studies have been carried out in the context of SBT. To the author's knowledge, no other empirical studies have been conducted on this particular topic.

Nevertheless, the desire of reducing test-suite sizes has received quite a lot of attention. Though based on other ideas for reducing the test suites, Heimdahl and George [29] found that the size of the specification based test-suites can be dramatically reduced and that the fault detection of the reduced test-suites is adversely affected. Wong *et al.* [30] investigated the effect on fault-detection of keeping block and all-uses coverage constant while reducing the size of a test suite. They found that effectiveness reduction was not significant even for the most difficult faults, which suggests that minimization of test suites can reduce the cost of testing at slightly reduced fault-detection effectiveness.

From the perspective of executing MBT in practice with respect to limited time and resources, three papers on similarity-based test selection address the problem of large test suites that are automatically generated by MBT-tools. Addressing the topic of scalability with respect to large test-suite sizes when applying model-based testing in practice, Hemmati and Briand [4] investigated and compared possible similarity functions to support similarity-based test selection. Empirical data on the most cost-effective similarity measure was collected by applying the proposed similarity measures and a selection strategy to an industrial software system. Results from the case study showed that using Jaccard Index to measure the similarity of the test cases (which were represented as a set of trigger-guards) of the respective test paths obtained the best results in terms of cost and effectiveness. They reported a significant reduction (77 percent) in test execution cost.

Continuing the work presented in [4], but this time trying to gain insights into why and under which circumstances a particular similarity-based selection technique can be expected to work, Hemmati *et al.* [5] investigated the properties of test suites with respect to similarities among fault revealing test cases. They conducted experiments based on simulation where two industrial case studies were used to guide the simulations. Obtained results confirmed their assumptions about similarity-based test case selection would perform better when "test cases which detect distinct faults are dissimilar and test cases that detect a common

fault are similar". They also found that similarity-based test case selection is less effective in cases when a small group of transition paths is mostly disconnected from the rest of the state machine.

Having a motivation similar to Hemmati and Briand, Cartaxo *et al.* [6] also addressed the problem of large test suites. A test-case selection strategy was compared with random selection by considering transition-based and fault-based coverage. Based on results from three case studies, they found that the similarity-based test case selection can provide more effective test suites than random selection.

Also, several studies address approaches that identify the differences between versions of a model for the purpose of reducing the sizes of regression test suites.

10.6 Discussion

There is a trade-off between a sufficiently detailed level in the test model and the quality of the resulting test cases. Reduced costs of generating tests may on the other hand increase the number of undetected faults. Removing contents from super states resulted in significantly smaller test suites, reducing the costs, yet retaining its fault-detection ability at a reasonable level. As reported in the results, however, a large part of the generated test cases were infeasible due to guard conditions that could not be satisfied. The reason for this being sub-state specific values that could not be controlled in the same way as when these sub states were included in the test model. To increase the number of feasible tests, ulterior work is necessary with respect to test-data selection. Although our results on using abstract test models as input to test generation proved acceptable fault-detection effectiveness combined with certain strategies, we must take into account the omitted details and be aware of those parts that cannot be tested based on the model [36]. We found that mutants that were seeded in the removed sub states were not detected.

An interesting observation is that by removing details from the test model and using a stronger criterion, results show that a comparable cost-effectiveness was obtained as compared to test suites generated from the complete test model.

Although generated based on a different idea, our results support the findings of Wong *et al.* [30] that test set minimization can greatly reduce the evaluation costs, and thus the cost of testing, at very little loss in fault-detection effectiveness. This is however, dependent on the choice of strategy.

10.7 Summary – Cost-Effectiveness for Complete Test Model versus Abstract Test Model

This chapter addressed whether or not varying the level of details in test models affected the cost-effectiveness of the state-based coverage criteria all transitions (AT), all round-trip paths (RTP), all transition pairs (ATP), paths of length 2 (LN2), paths of length 3 (LN3) and paths of length 4 (LN4). Two different oracles were applied.

Section 10.1 presented descriptive statistics for the obtained results on cost and effectiveness. We saw that the test-suite sizes significantly decreased when abstracting the test model. Infeasible test cases were introduced by the abstract test model due to the sub states being removed. This implied less control over test data with respect to externally controlled variables.

For oracle O1, the greatest reduction in test-suite size was seen for AT (from 166 to 17 test cases), closely followed by ATP (1425 to 192 test cases). RTP was reduced from 299 to 66 test cases. The LN4 test suite was reduced from 764 to 415 test cases. The remaining strategies were not affected to the same extent; LN2 and LN3 were reduced, by 14.8 and 29.4 percent, respectively. By using oracle O2, the reduction was overall lesser due to lower number of infeasible test cases as compared to using oracle O1, but quite consistent with the results for O1.

Observations showed that it takes significantly shorter time for all strategies to prepare test suites from the abstract test model as compared to the complete test model. The greatest difference was seen for ATP (96 percent for both O1 and O2), closely followed by AT (95 percent for O1 and 94 percent for O2). RTP was reduced by 62 percent for O1 and 64 percent for O2. Also LN2 and LN3 were prepared in less time from the abstract model – LN2: 34 percent for O1 and 33 percent for O2; LN3: 38 percent for O1 and 40 percent for O2. Slightly less difference was seen in the results for LN4: 26 percent and 27 percent for O1 and O2.

We saw a similar trend for execution time with great differences between the times spent on executing the test suites generated from the abstract model as compared to the complete model. The largest reduction was achieved by AT (97 percent for O1 and 96 percent for O2), followed by ATP (88 percent for O1 and 80 percent for O2), and RTP (85 percent for O1 and 77 percent for O2). When O1 was combined with the abstract model for LN2, LN3, and LN4, results show that the preparation times were reduced by 11 percent, 38 percent, and 22 percent, respectively. No difference was found between the abstract and complete model when applying O2 to LN2. A minor reduction was found for LN3 (4 percent). The execution time for LN4, on the other hand, was increased by 6 percent.

Regarding LN2, LN3, and LN4, we see an overall smaller difference between the two models. This, however, can be explained by the first surrogate measure of cost, test-suite size. Recall from Table 29 that significantly smaller test-suite sizes were found for the abstract model versus the complete model for LN2, LN3, and LN4 contra AT, RTP, and ATP. Thus, lower differences in execution and preparation time would be expected results.

Comparing the two oracles applied to the abstract test model provided rather consistent results as compared to what was seen for the complete test model. Differences in preparation time for the abstract model was in the range from -5 percent to 2 percent compared to -7 percent to 4 percent for the complete model. Regarding execution time, we saw a similar correlation between the abstract and the complete model: for the abstract model, test suites combined with O2 spent from 68 percent to 74 percent less time on execution compared to O1, whereas test suites generated from the complete model combined with O2 spent 72 percent to 83 percent less time on execution compared to O1.

An overall trend in the results for mutation score was that the abstract test models obtained lower mutation score means than what were achieved by the complete test models.

Section 10.2 regarded statistical tests. The non-paired Wilcoxon signed-rank test was applied to the replicated data, i.e., for AT, RTP, and ATP. Overall, i.e., considering the three strategies combined with each of O1 and O2, significant differences were found for each of preparation time, execution time and mutation score. Significance was found for $\alpha = 0.01$.

Test suites generated from the complete model required both higher preparation and execution time (large effect sizes were found for both measures). On the other hand, the complete models achieved higher mutation scores.

To summarize, the results showed that ATP applied to a detailed model is an expensive strategy. The high fault-detection effectiveness may be at a too high cost. When combined with the abstract model, cost was significantly reduced. The effectiveness was also reduced, but not as much as the cost. On the other extreme, LN2 had the lowest cost but also the lowest effect; at least for the complete model. Results for the abstract model combined with oracle O1 showed similar results to what was found for AT applied to the abstract model. The level of details in the model had an enormous impact on the cost and effectiveness for AT; all mutants were killed by AT when using the complete model, although at a large increase in cost. The complete model combined with oracle O1 obtained as good mutation score as ATP and AT also for RTP and LN4. Of these, RTP had the lowest costs. Overall, the abstract model performs better with oracle O1.

AT, RTP, ATP, and LN4 all provided the highest mutation scores when generated from the complete model used with oracle O1. Of these, RTP had the lowest costs. Using the weaker oracle O2, still based on the complete test model, the effectiveness was slightly reduced: AT, RTP, and LN4 killed 80 % of the mutants. Regarding the abstract test model, we saw that LN3, LN4, RTP, and ATP killed 87 % of the mutants – interestingly, the cost of ATP was dramatically reduced as compared to the test suites generated from the complete model.

Conclusions: Reducing the level of detail in the test model significantly influences the cost-effectiveness. Results show that both costs and fault-detection ability are lower for test suites generated from the abstract model as compared to the complete model.

In the next chapter, we will see the results from applying sneak-path testing.

11 Case Study 4 – What is the Impact of Sneak-Path Testing on the Cost-Effectiveness?

Chapters 8–10 focused on conformance testing, aiming at detecting deviations from specified system behaviour when *expected* events were invoked on the SUT. The case study presented in this chapter, on the other hand, presents the results from augmenting the conformance testing with sneak-path testing. In contrast to conformance testing, sneak-path testing [31] feeds the SUT with unexpected events that should not trigger any change of state in the SUT. For each state in the SUT, all possible events that are not specified for the particular state are invoked. This technique is intended to catch faults that introduce undesired, additional behaviour, in terms of extra transitions and actions. In this study, we aimed to investigate how sneak-path testing affects the cost of testing and its fault detection rates.

RQ4: What is the impact of sneak-path testing on the cost-effectiveness?

Section 11.1 and Section 11.2 present cost and effectiveness, respectively. Results are discussed in Section 11.3.

11.1 Cost

This section presents the collected data on cost – the test-suite sizes and the time spent on preparing and executing the sneak-path test suites.

11.1.1 Test-Suite Size

The size of the sneak-path test suite is equal to the number of states in the SUT, and hence is low compared to the other testing strategies previously addressed in this thesis. The length of each test case depends on two matters: (1) the length of the path that must be traversed in order to reach the particular state to be tested, and (2) the number of known unexpected events for the state.

A Kermeta-transformation was used to generate 68 separate state machines from the complete test model. Each state machine included a path from the initial state to the state that was tested, and also the unexpected events for that state. The complete transformation took 2,460 seconds.

Due to limited time, the aim of sneak-path testing in this study was to show that the mutants actually could be killed – not to show *how* many of the test cases that could kill each of the mutants. Therefore, it was decided to select a subset of the test cases which were

expected to kill the mutants. From analysis of the total of 68 state machines, 52 were affected by the seeded sneak paths. Furthermore, only seven distinct single states among the 52 combined states were directly involved in the seeded sneak paths. Thus, seven state machines that contained the distinct affected states were included in the sneak-path testing. That is, the size of the complete sneak-path test suite was 68 test cases – only seven, however, were executed. Table 43 shows the selected state machines.

Table 43 Selected state machines – complete test model

State machine ID	Test case from state machine
4	ExtraSlowConfirmInit
12	ManualFSConfirmSoftHalt
16	AutoConfirmEvaluateHalt
15	ExtraSlowConfirmedDEFinal
7	PreManualFSDisabled
18	ManualFSConfirmedSoftStop
8	AutoConfirmedEvaluateHalt

The same strategy as for the complete model was used to make a selection of test cases from the sneak-path test suite for the abstract model. The Kermeta transformation spent 116 seconds on generating 21 state machines – one state machine for each state in the abstract model. Table 44 shows the selected state machines that were affected by the seeded sneak paths. Again, the test suite consisted of 21 test cases of which three were executed on the mutants.

Table 44 Selected state machines – abstract test model

State machine ID	Test case from state machine
3	ManualFSInit
5	AutoEnabled
7	ExtraSlowInit

11.1.2 Time

Table 45 shows the time spent on preparing the sneak-path test suite for the complete model. Please note that each test case was run separately. The time varies from 23 seconds to 34 seconds when using oracle O1, and from 21 seconds to 34 seconds when using oracle O2.

Table 45 also displays the execution time observed for oracles O1 and O2 when combined with the complete test model. The minimum and maximum times for oracle O1 were, respectively, three and seven seconds. Compared to oracle O1, we see an overall reduction in execution time by applying oracle O2 – each test case spent two seconds on execution when applying O2.

Results for the abstract test model regarding preparation time reveals little distinction between the two oracles. The minimum time for oracle O1 was 18 seconds, whereas the maximum measured preparation time was 21 seconds. Regarding O2, we see values from 17 to 18 seconds. Similar was observed for the execution time. Although the differences between the two oracles vary from 33 percent to 60 percent, the actual values are not so different – oracle O1 spent three to five seconds on executing the test cases, whereas O2 spent two seconds.

Table 45 Preparation time and execution time – complete test model

State machine ID	Test case generated from state machine	Oracle	TIME PREPARE TEST CASE (sec)	Diff O2 by O1 (%)	TIME EXECUTE TEST CASE (sec)	Diff O2 by O1 (%)	Executed on Mutant
7	PreManualFS	O1	25	0.0	7	-71.4	M23
		O2	25		2		
12	ManualFSConfirm	O1	26	-11.5	4	-50.0	M18
		O2	23		2		
18	ManualFSConfirmed	O1	26	0.0	4	-50.0	M24
		O2	26		2		
16	AutoConfirm	O1	25	-12.0	6	-66.7	M19
		O2	22		2		
8	AutoConfirmed	O1	25	-12.0	4	-50.0	M25
		O2	22		2		
4	ExtraSlowConfirm	O1	28	-25.0	3	-33.3	M16
		O2	21		2		
4	ExtraSlowConfirm	O1	25	-16.0	4	-50.0	M17
		O2	21		2		
4	ExtraSlowConfirm	O1	23	-8.7	5	-60.0	M20
		O2	21		2		
15	ExtraSlowConfirmed	O1	34	0.0	6	-66.7	M21
		O2	34		2		
15	ExtraSlowConfirmed	O1	34	0.0	4	-50.0	M22
		O2	34		2		
15	ExtraSlowConfirmed	O1	34	0.0	4	-50.0	M26
		O2	34		2		
	TOTAL TIME O1 (sec)		305		51		
	TOTAL TIME O2 (sec)		283		22		

Table 46 Preparation time and execution time – abstract test model

State machine ID	Test case from state machine	Oracle	TIME PREPARE TEST CASE (sec)	Diff O2 by O1 (%)	TIME EXECUTE TEST CASE (sec)	Diff O2 by O1 (%)	Executed on Mutant
3	ManualFSEnabled	O1	19	-10.5	4	-50.0	M18
		O2	17		2		
3	ManualFSEnabled	O1	19	-5.3	4	-50.0	M23
		O2	18		2		
3	ManualFSEnabled	O1	19	-10.5	4	-50.0	M24
		O2	17		2		
5	AutoEnabled	O1	21	-14.3	4	-50.0	M19
		O2	18		2		
5	AutoEnabled	O1	19	-10.5	3	-33.3	M25
		O2	17		2		
7	ExtraSlowInit	O1	19	-10.5	5	-60.0	M16
		O2	17		2		
7	ExtraSlowInit	O1	20	-10.0	4	-50.0	M17
		O2	18		2		
7	ExtraSlowInit	O1	20	-15.0	4	-50.0	M20
		O2	17		2		
7	ExtraSlowInit	O1	18	-5.6	4	-50.0	M21
		O2	17		2		
7	ExtraSlowInit	O1	19	-10.5	5	-60.0	M22
		O2	17		2		
7	ExtraSlowInit	O1	20	-10.0	5	-60.0	M26
		O2	18		2		
	TOTAL TIME O1 (sec)		213		46		
	TOTAL TIME O2 (sec)		191		22		

In practice, the complete sneak-path test suite is required to be run as, of course, the location is not know. As we did not run the complete sneak-path test suite, the cost of full execution cannot be provided; however, estimates based on the collected data are presented in Table 47. When using oracle O1, the collected data shows that preparing the sneak-path tests from the complete model took 305 seconds, whereas execution of the tests took 51 seconds. Realistic, but rough, estimates for the *total* sneak-path test suite are 1,885 seconds ((305 ÷ 11) × 68) in preparation time and 315 seconds ((51 ÷ 11) × 68) in execution time.

For the abstract test model, the collected data shows that preparing the sneak-path tests took 213 seconds, whereas execution of the tests took 46 seconds. Realistic, but rough, estimates for the *total* sneak-path test suite are 407 seconds ((213 ÷ 11) × 21) in preparation time and 88 seconds ((46 ÷ 11) × 21) in execution time.

Using oracle O2 when applying sneak-path tests on the complete model resulted in the following cost: preparing the sneak-path tests took 283 seconds, whereas execution of the tests took 22 seconds. Realistic, but rough, estimates for the *total* sneak-path test suite are 1,750 seconds ((283 ÷ 11) × 68) in preparation time and 136 seconds ((22 ÷ 11) × 68) in execution time.

Preparation time for the abstract test model applied with oracle O2 was 191 seconds; execution time was 22 seconds. Realistic, but rough, estimates for the *total* sneak-path test suite are 365 seconds ((191 ÷ 11) × 21) in preparation time and 42 seconds ((22 ÷ 11) × 21) in execution time.

Table 47 Estimates preparation and execution time for sneak-path test suites

		Preparation time (estimat)	Execution time (estimat)
Complete test model	**Oracle O1**	1,885	315
	Oracle O2	1,750	136
Abstract test model	**Oracle O1**	407	88
	Oracle O2	365	42

11.2 Effectiveness

Table 48 shows the observed results on effectiveness for the complete model, both for oracle O1 and O2. Augmenting the conformance test suites with sneak-path testing generated from the complete model resulted in the remaining mutants being killed. Each and every mutant that consisted of additional implemented behaviour beyond specified behaviour was killed by sneak-path tests.

Table 49 presents the results on killing mutants with sneak-path tests from the abstract model. Recall that the sub states in the super states were removed from the abstract model. This implies less control over the testing process; moreover, removing the sub states particularly regards external variables and infeasible state combinations that may be the reason for why test cases fail prior to killing the mutant.

The sneak-path test suite generated from the abstract model killed 10/11 sneak paths. The test case from state machine 5 that did not kill the expected mutant (M19) failed due to an infeasible step in the test case. This test case failed because of a guard on the transition from the `Initial` state to the `Enabled` state that was not satisfied. The transition required the variable `blockdriveenable` to be false. As the current mode sub state was `AutoConfirm`, the `blockDriveEnable` was true and thus in the wrong sub state.

Table 48 Results from killing mutants with sneak-path testing – complete test model

State machine ID	Test case from state machine	From state	To state	Mutant	Oracle	Killed by operation
4	ExtraSlowConfirmInit	ExtraSlowConfirm	AutoConfirm	M16	O1	WriteMode(Auto)
					O2	WriteMode(Auto)
		ExtraSlowConfirm	ManualFSConfirm	M17	O1	WriteMode(ManualFS)
					O2	WriteMode(ManualFS)
		ExtraSlowConfirm	AutoState	M20	O1	StartInAuto()
					O2	StartInAuto()
12	ManualFSConfirmSoftHalt	ManualFSConfirm	ExtraSlowConfirm	M18	O1	WriteMode(ExtraSlowSpeed)
					O2	WriteMode(ExtraSlowSpeed)
16	AutoConfirmEvaluateHalt	AutoConfirm	ExtraSlowConfirm	M19	O1	WriteMode(ExtraSlowSpeed)
					O2	WriteMode(ExtraSlowSpeed)
15	ExtraSlowConfirmedDEFinal	ExtraSlowConfirmed	AutoConfirm	M21	O1	WriteModeAuto()
					O2	WriteModeAuto()
		ExtraSlowConfirmed	ManualFSConfirm	M22	O1	WriteMode(ManualFS)
					O2	WriteMode(ManualFS)
		ExtraSlowConfirmed	AutoState	M26	O1	StartInAuto()
					O2	StartInAuto()
7	PreManualFSDisabled	PreManualFS	ExtraSlowConfirm	M23	O1	WriteMode(ExtraSlowSpeed)
					O2	WriteMode(ExtraSlowSpeed)
18	ManualFSConfirmedSoftStop	ManualFSConfirmed	ExtraSlowConfirm	M24	O1	WriteMode(ExtraSlowSpeed)
					O2	WriteMode(ExtraSlowSpeed)
8	AutoConfirmedEvaluateHalt	AutoConfirmed	ExtraSlowConfirm	M25	O1	WriteMode(ExtraSlowSpeed)
					O2	WriteMode(ExtraSlowSpeed)

Table 49 Results from killing mutants with sneak-path testing – abstract test model

State machine ID	Test case from state machine	From state	To state	Mutant	Oracle	Killed by operation
3	ManualFSEnabled	ManualFS	ExtraSlow	M18	O1	WriteMode(ExtraSlowSpeed)
					O2	WriteMode(ExtraSlowSpeed)
		ManualFS	ExtraSlow	M23	O1	WriteMode(ExtraSlowSpeed)
					O2	WriteMode(ExtraSlowSpeed)
		ManualFS	ExtraSlow	M24	O1	WriteMode(ExtraSlowSpeed)
					O2	WriteMode(ExtraSlowSpeed)
5	AutoEnabled	Auto	ExtraSlow	M19	O1	N/A - infeasible test case
					O2	N/A - infeasible test case
		Auto	ExtraSlow	M25	O1	WriteMode(ExtraSlowSpeed)
					O2	WriteMode(ExtraSlowSpeed)
7	ExtraSlowInit	ExtraSlow	Auto	M16	O1	WriteMode(Auto)
					O2	WriteMode(Auto)
		ExtraSlow	ManualFS	M17	O1	WriteMode(ManualFS)
					O2	WriteMode(ManualFS)
		ExtraSlow	Auto	M20	O1	StartInAuto()
					O2	StartInAuto()
		ExtraSlow	Auto	M21	O1	WriteModeAuto()
					O2	WriteModeAuto()
		ExtraSlow	ManualFS	M22	O1	WriteMode(ManualFS)
					O2	WriteMode(ManualFS)
		ExtraSlow	Auto	M26	O1	StartInAuto()
					O2	StartInAuto()

11.3 Summary

In this chapter, sneak-path testing [31] was applied to the SUT. Recall that 11 of 26 seeded faults were sneak-paths.

Complementing state-based testing (Chapter 8–10) with sneak-path testing at an additional cost in preparation and execution time resulted in that the remaining 11 mutants were killed – those 11 mutants were not killed by any of the six state-based coverage criteria. Execution of the sneak-path test suite on the abstract model killed 10/11 sneak-paths. This was, however, due to an infeasible test case.

Being equal to the number of states in the SUT, the cost of sneak-path test suites are rather inexpensive as compared to the state-based coverage criteria investigated in this study. Recall that the length of each test case depends on two matters: (1) the length of the path that must be traversed in order to reach the particular state to be tested, and (2) the number of known unexpected events for the state.

The results presented in this chapter demonstrate that the testing strategies are complementary in order to catch different types of faults. Thus, the results indicate quite strongly that *sneak-path testing is a necessary step in state-based testing due to the following observations: (1) the proportion of sneak paths in the collected fault data was high (42 %), and (2) the presence of sneak paths is undetectable by conformance testing.*

Our results support the recommendation of Binder [31] and the conclusions drawn in the study of Mouchawrab *et al.* [92]: Testing sneak paths is an essential component of state-based testing in practice. The additional cost is justified by the positive influence on fault-detection effectiveness.

Conclusions: The obtained results confirm the importance of including sneak-path testing to improve fault-detection effectiveness of state-based testing, and strongly indicate that sneak-path testing is a necessary step in state-based testing.

The next part (Part IV) will now present lessons learned, threats to validity, conclusions, and, finally, future work.

PART IV – SUMMARY

Introduction

In this final part (Part IV), we will read about challenges and likely benefits of TRUST in particular and SBT in general, presented as lessons learned in Chapter 12. Threats to validity are discussed in Chapter 13. Conclusions of this thesis are presented in Chapter 14. Finally, Chapter 15 addresses future work.

12 Lessons Learned

This chapter reports the lessons learned from modeling to facilitate automatic testing, and from execution of the automatically generated test suites.

12.1 Remove Illegal State Combinations and Infeasible Transitions from the Flattened State Machine

We experienced that numerous test cases failed due to infeasible paths. Infeasible test cases exist in the generated test suites because of guards on certain paths that cannot be satisfied. There are mainly two reasons for why the guards on the illegal test cases cannot be satisfied: (1) due to illegal state combinations, and (2) because of wrong test data.

One example can be given from the ATP test suite used with the weakest oracle O2 generated from the abstract test model: An infeasible transition from state `AutoEnabled` to state `AutoDisabled` triggered by the event `ClearEnableDevice` guarded by [`mode<>A and modeSoftStop = false`]**.** This particular transition is infeasible because of the guard which requires that the mode is not Auto.

Lesson learned: Ensure that all illegal state combinations are removed from a flattened state machine. Moreover, closely inspect the flattened state machine to ensure that no transition exists whose guard is in conflict with the source state of the transition.

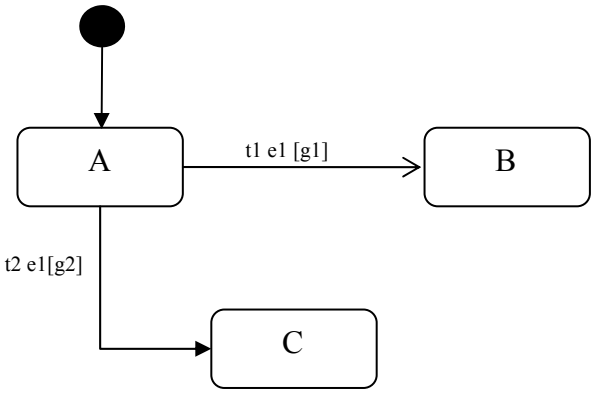
12.2 Modeling and Coding to Facilitate Automation of State-Based Testing

Certain modeling aspects can be challenging to state-based testing. This section addresses examples of such aspects.

Example 1: Figure 50 shows a state machine consisting of transition `t1`, sourced in state `A` and targeted in state `B`; and transition `t2`, sourced in state `A` and targeted in state `C`. Both transitions are triggered by event `e1`. Transition `t1` is guarded by `g1`, whereas `t2` is guarded by `g2`. A typical state-based test case `TCa` would put the SUT in state `A`, invoke the event `e1`, and expect the SUT to transition to state `B`, if the guard `g1` is satisfied. Another test case `TCb` would put the SUT in state `A`, invoke event `e1`, and then expect the SUT to transition to state `C`, if guard `g2` is satisfied. However, if the guards are mutually exclusive, we have a problem as the SUT will transition in both test cases independent of the evaluation of the guards. Let us say that guard `g1` contains the following expression: `x = true`, and guard `g2` contains: `x = false`. This means that test case `TCa` will fail if `x = false` because the SUT will transition to state `C`. This is correct behavior of the SUT, but will, however, cause erroneous

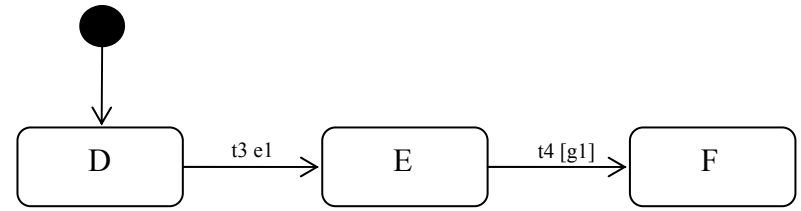
failure of test cases. Hence, in order to successfully test the two transitions, it is necessary to provide the necessary test data to force a particular path according to the test case.

Figure 50 Example 1 – Modeling Challenges related to Automatic Test Case Generation



Example 2: Another modelling aspect that can cause a test case to fail even though the SUT behaves correctly is the use of a particular type of event; the completion event (i.e., events that are the completion of state behaviour). The state machine presented in Figure 51 illustrates a situation where a transition is triggered by the completion event from state `E`. Test case `TCc` checks if the SUT transitions to state `E` from state `D` via transition t3. Now, if state `E` has an outgoing transition `t4` targeted in state `F` with no explicit event other than the completion event, the SUT will immediately transition to `F` unless state `E` has state behaviour. In this situation, `TCc` will fail because the resulting state is `F` and not the expected state `E`. This means that in order for `TCc` to pass, the test data must be forced to *not* satisfy `g1` at the point state `E` is reached and exit of the state is triggered by the completion event.

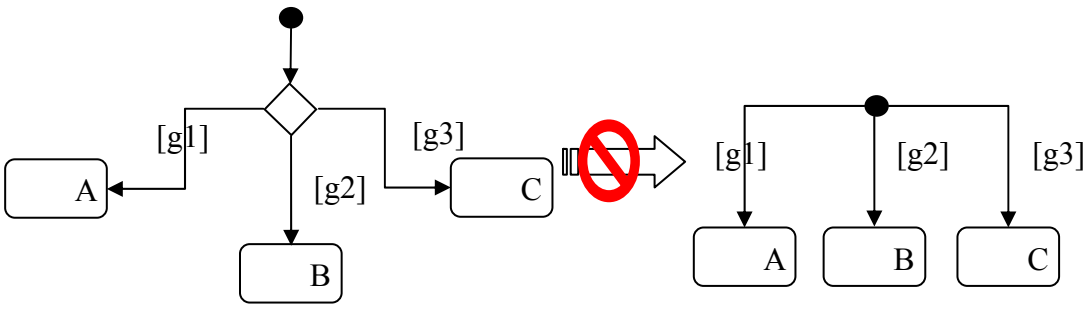
Figure 51 Example 2 – Modeling Challenges related to Automatic Test Case Generation



Example 3: The simple transition type and the compound transition type are not always separated when being referred to in the UML 2.0 specification. This may cause confusion for instance in relation to the initial pseudostate connected to choice points as the initial state is defined to have only one outgoing transition. TRUST did not support test-case generation from state machines having choice points, and as the UML 2.0 specification [124] was to some extent vague about the different types of transitions, several questions regarding outgoing transitions from initial state were raised. Unfortunately, the flattening component of TRUST wrongly resolved, as illustrated in Figure 52, choice points by redirecting each outgoing transition from a choice point to the target of the incoming transition to that choice point.

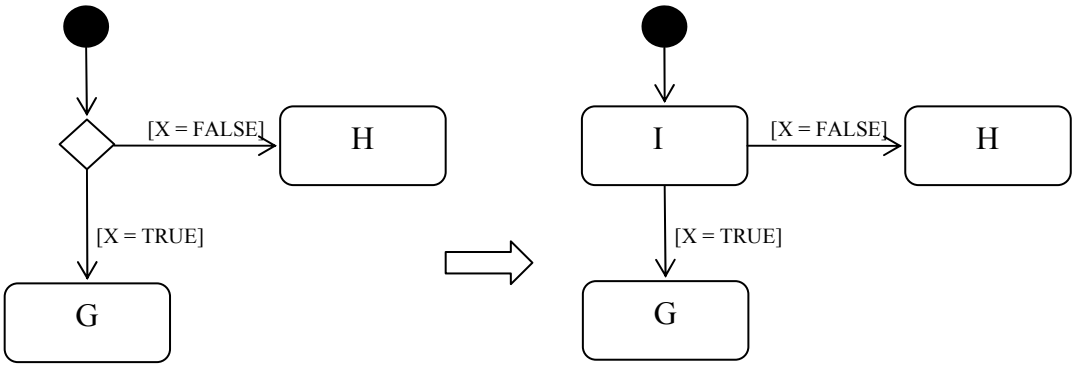
This misunderstanding was, however, clarified with Selic from the OMG group who stated the following: "A transition in UML is a directed arc between two nodes in the graph – whether or not the nodes are proper states or pseuedostates. So, the arc between the initial state and the choice point is a fully-fledged transition. A path from the initial psuedostate through the choice point and to a state is known as a compound transition."

Figure 52 Example of Wrong Interpretation of the UML 2.0 Superstructure



In our project, all these cases were re-modelled by adding a new state without behaviour to replace the choice point as shown in Figure 53.

Figure 53 Example 3 – Modeling Challenges related to Automatic Test Case Generation



Lesson learned: Ensure that the required test data are correctly selected not only in prior to invoking the event as to enable firing of the transition (Example 1), but also at the point the target state has been reached (Example 2) in order to prevent the next transition to be fired and thus to keep the system in the target state of a test case. Finally, we experienced the UML 2.0 specification [124] to be vague in the description of the different types of transitions (example 3). As the tool we applied did not support test case generation from state machines including choice points, one of the questions we had was regarding how to remodel and resolve this kind of construct. As a choice point may not be resolved by merging the incoming and outgoing transition to and from the choice point, our best solution was to replace the choice point with a new state without behaviour.

12.3 Improving the Model/Code through Iterative State-Based Testing

The general idea in model-based testing is to have a model of the SUT that represents its intended behavior. This model should be at a higher abstraction level than the SUT. A test model with less precision eases its validation according to SUT behavior before generating tests from it. A trade-off in model-based testing is thus to find the balance between the test model abstraction level and the fault-detection ability of the test suites generated from this model [36].

In this study, however, the test model of the SUT was in fact a precise model of the SUT behavior. This provided us with test suites that capture the behavior of the SUT at a detailed level. During initial testing of the SUT, results showed that several test cases failed. We found that the flattened model of the SUT contained errors. Analyzing test cases that failed helped

us to identify illegal state combinations, infeasible transitions, and transitions with incomplete guard expressions.

For our case studies, precise behavioral modeling of complex industrial systems using standard UML 2.0 state machines was a prerequisite for using TRUST. The flattening component requires that it is provided with a correctly specified state machine and currently does not provide any feedback in case of errors in the model. Modeling correctly, however, is not a trivial task and requires that the UML specification be carefully studied. Even though constructs like concurrency and hierarchy are supposed to ease the understandability of large state machines, such constructs may actually confuse the developer. In particular, we experienced that concurrency, if not carefully applied, could introduce modeling errors in practice. For example, concurrent regions sometimes make it difficult to see the set of transitions between state combinations. A typical fault is that a guard is missing on a transition, which allows for transitions to state combinations that are illegal targets from particular source states. However, we found that it helped to inspect the flattened state machine to detect such mistakes.

Lesson learned: We experienced that iterative state-based testing can contribute in developing models and code of higher quality in that testing the code based on the model detects inconsistencies between model and code.

12.4 Test Results provided by the Oracle

Existing research do not report experiences on execution details. This section provides information regarding the usefulness of the provided information provided by the implementation of our oracles.

The following information was provided as a result of how we implemented the oracles: (1) the test case number, (2) whether the test case passed of failed, (3) if the test case failed, the operation that implemented the event on the transition that caused the test case to fail was reported, and (4) if the test case failed, the reason for why the test case failed – that is, either violation of the state invariant or the SUT being in the wrong state.

The reported information eased the analysis of the un-killed mutants in that the relevant test case could be examined. Moreover, as the operation that caused the test case to fail was explicitly stated, the specific position in the test case could be quickly identified. We used this information to stop the re-run of the execution at that particular place in the code to identify the reason for why the test case failed.

Lesson learned: Reporting the particular test case number, the class where the fault occurred, and the specific operation that caused the test case to fail, was experienced as highly useful in the analysis of the test cases.

12.5 Practical Issues – Visual Studio caused Re-Run of Test-Suites

Executing the test suites in Microsoft Visual Studio 2008 in combination with the Java Virtual Machine was extremely time-consuming as the results were only partially reported. We had to re-execute the test suites numerous times in order to collect complete test results.

Lesson learned: JVM and Microsoft Visual Studio 2008 is not an optimal combination. Other environments for executing test drivers should be explored.

13 Validity

Empirical studies of cost-effectiveness introduce validity concerns. It is thus important to identify possible threats to validity to increase the awareness regarding this particular matter when interpreting the results. Furthermore, prospective replications of the study may then alter the study design in order to address problematic threats. The subsequent sections discuss what the author of this thesis considers as threats to the validity.

13.1 External Validity

External validity concerns to what extent it is possible to generalize the findings, and to what extent the findings are of interest to other people outside the investigated case [122], i.e., to whether a domain exists to which the results are relevant. But how is it possible to generalize from a single case? Yin [64, p. 10] provides the following answer:

> *The short answer is that case studies, like experiments, are generalizable to theoretical propositions and not to populations or universes. In this sense, the case study, like the experiment, does not represent a "sample", and in doing a case study, your goal will be to expand and generalize theories (analytic generalization) and not to enumerate frequencies (statistical generalization).*

As stated by Runeson and Höst [122], for case studies, the intention is to enable analytical generalization where the results are extended to cases which have common characteristics and hence for which the findings are relevant, i.e., defining a theory.

The main strength of this study is, in fact, its external validity. Two factors in particular increase the external validity, namely the industrial context and the use of real faults when evaluating the testing strategies. The system in focus of this thesis is highly representative of control systems with state-based behavior thus improving the external validity. It is important to provide detailed context descriptions, like system characteristic, development and testing procedures, such that others can relate the results to their own context. This information was provided in 7.2. Moreover, in contrast to the majority of existing studies, applying artificial faults, the faults used in this study are real faults collected in a field study conducted in ABB (Section 7.2.7.2). In spite of these two factors, however, there are several issues that should be discussed.

First of all, let us consider the SUT. As stated by Briand *et al.* [22]: "Results will always, by definition, be specific to the SUT". An obvious threat to the external validity of this study, which reduces its potential for contributing with general results, is the fact that only one system was used in the evaluation. Even though it is a highly appropriate and relevant case, as it was developed by an external industry group; it is large, it represents a real system, and it is of real world importance, drawing general conclusions is not possible from this one case. However, although the obtained results are specific to the SUT, it is an example of a typical control system. The characteristics of the selected system are expected to be similar for many control systems, which may increase possibility of these results to be valid for these types of systems. The SUT was entirely modeled using Boolean and enumerated variables, which most certainly affects obtained results and makes it impossible to generalize the results to systems that, for example, contain numeric variables and constraints. This implies that the results may not be valid for control systems that make use of other types of variables.

In previous studies where artificial mutation operators have been applied in the evaluation of testing strategies, the question of its impact on the external validity has been raised. The use of real faults when generating mutant programs is not common practice. In this study, however, we purely used real faults to generate mutants. Although only 26 mutants were applied in this study, the seeded faults were real and manually extracted from a field study as described in Chapter 7. But again, it is not certain that the results apply to other organisations as it depends on the developer what kind of faults that are introduced to a system. Hence, further studies are necessary to increase the external validity of the results. Having the preparation and execution time in mind, the feasibility of the study will be threatened by a dramatic increase in number of mutants. To avoid masking of faults, only one fault was seeded per mutant program. Thus, like studies presented in related work (see Table 1), this

study only evaluates the detection of single faults. Complex fault patterns have not been studied in this thesis.

Also note that the SUT was developed in C++. The results may vary for other programming languages.

Nevertheless, the results are believed to be representative for control systems and can be used to guide readers when selecting testing strategies. As for any empirical studies, however, this study should be replicated for other types of faults and other control systems in order to make the results stronger and prove the results relevant for other similar projects.

13.2 Internal Validity

Internal validity is "of concern when causal relations are examined" [122]. Investigated variables may also be affected by extraneous variables, confounding factors, not accounted for in the study, thus a threat to the internal validity. What is observed should be attributed to the studied variable and not to potential confounding factors. Internal validity thus concerns situations where the researcher examines the influence of factor A on factor B, when in fact factor C also has an impact on factor B. Threats to validity occur when the researcher is not aware of factor C, but concludes that factor A caused factor B.

A threat to internal validity is the fact that the author of this thesis was the subject in the case studies. The rationale for this being lack of resources and a general lack of state-based testing experience in the company. Also important to notice is that the purpose of the case studies was to demonstrate the possible gains ABB could retrieve by applying state-based testing in the future. That is, the main purpose was *not* to study the interaction between the tester and the technology – the focus was directed towards the actual achievements that could be obtained with respect to cost-effectiveness by introducing state-based and sneak-path testing. The industry needs help in introducing new techniques, and this is one pragmatic approach in order to demonstrate possible advantages of this particular technique.

One detected risk in terms of internal validity was the possible randomness in the obtained results for three of the coverage criteria AT, RTP, and ATP. This issue was handled by generating 30 test trees for those coverage criteria, thus replicating the experiment for these criteria 30 times. Statistical hypothesis testing was applied to the collected data to identify whether or not significant differences were present between the three criteria.

Another threat to internal validity in this study could be related to the fault-detection rate due to the fact that both the generation of test cases and the insertion of faults to develop

mutants were conducted by the author. Because of this, the author could potentially influence the implementation of the test suites as to be better at detecting certain types of faults. Ideally, generation of test cases and fault seeding should be conducted by different people. Nevertheless, as the test suites were automatically generated by following known algorithms, this is not considered being a threat – the implementation of the algorithms do not suffer a great risk of being manipulated in favour of detecting specific faults and should be independent of the person who implements it. Furthermore, the seeded faults were actual faults introduced by developers from ABB. Hence, the author had no impact on the seeded faults.

The final threat to internal validity, that we are aware of, is related to the fault-detection ability due to the problem of infeasible test cases. This particularly regards test suites generated from the abstract test model due to little control over internal states. In future studies, more advanced test case selection must be applied in order to remove infeasible test cases, and moreover, ensure that the coverage is retained after removing infeasible test cases.

13.3 Construct Validity

Construct validity regards to what extent the operational measure that are studied really represent what the researcher have in mind and what is investigated according to the research questions [122]. That is, it concerns the establishment of correct operational measures for the concepts being studied. Wrong choices may affect the quality of the results.

Efforts were made to increase the construct validity in this study. Triangulation – the use of multiple data sources – helps to ensure construct validity. Several surrogate measures were selected to describe cost. Moreover, the selected measures are commonly known measures in cost-effectiveness studies in the field of software testing. In this study, like in similar experiments (e.g., [12]), cost and effectiveness were measured using surrogate measures. Cost was measured using not only test-suite size but also the time spent on preparing and executing the test suites. Effect was measured by the ability of the test suites in detecting faults.

However, as stated by Andrews *et al.* [28], test-suite size as a measure may fail to capture all dimensions of testing cost when considering different coverage criteria since it may be more difficult to find test cases that achieve some coverage criteria than others. In order to address this issue, selecting the most optimal test trees may reduce the validity threat. This was not done in this study, and hence, a validity threat can be non-representative figures for cost as compared to effect.

13.4 Reliability

Recall from Chapter 7 that reliability concerns to what extent the data and the analysis are dependent on the specific researchers [122], e.g., unclear descriptions of data collection procedures such that later replications of the study could give different results. This is addressed by providing as detailed design and description of analysis procedures as possible.

14 Conclusions

State-based testing based on UML models in industry is not yet a widespread practice. Even though the use of tools for execution of more intelligent testing is slowly increasing, common practice is often based on manual testing approaches. Moreover, to avoid the undesirable cohesion between the skills of a tester and the quality of the testing, the use of SBT in combination with proper tools facilitate sound, predictable testing that is somewhat independent of the individual tester.

Existing research has not yet reached a point where useful guidelines are developed as to advice organizations, not only on how and when to proceed with SBT, and what kind of SBT to apply, but also on how and when to combine SBT with other types of testing. More studies, to be suggested in Chapter 15, are essential prerequisites for creating such guidance to support practitioners when deciding upon which strategy to use. The findings in this thesis, however, contribute to such work by comparing SBT criteria under different oracle and test model conditions.

An empirical evaluation on the cost and fault-revealing capabilities of six state-based coverage criteria applied with two different oracles was presented in this thesis. The following coverage criteria were evaluated: all transitions (AT), all round-trip paths (RTP), all transition pairs (ATP), all paths of length two (LN2), all paths of length three (LN3), and all paths of length four (LN4). Moreover, the testing strategies were applied to both a precise model of the SUT and to less detailed model of the SUT where sub states and their belonging transitions were removed from super states.

The study was conducted in cooperation with the ABB Corporate Research Center in Norway (NOCRC) where a UML state-based technique was applied when developing the subsystem of a safety system for controlling a machine. The combinations of test coverage, oracle and model, as described in the previous paragraph, were then evaluated using mutation analysis with real faults extracted from a global field study at ABB, allowing us to compare the cost-effectiveness for the generated test suites. Test suites were automatically generated by using the model-based testing tool TRansformation-based tool for Uml-baSed Testing (TRUST). TRUST was developed motivated by the lack of extensible and configurable model-based testing tools. The software architecture and the implementation strategy of TRUST facilitate its customization to different contexts by supporting configurable and extensible features such as input models, test models, coverage criteria, test data generation strategies, and test-scripting languages.

Reported in this thesis, are results and experiences that may be helpful when selecting coverage criteria, type of oracle, and the level of details in the test model. As most studies evaluating coverage criteria are based on artificial mutation operators, this thesis contributes with its results from applying real faults on a module of a system from an industrial context. The subsequent sections summarize the results from each of the case studies presented in Chapter 8–11.

14.1 Case Study 1 – What is the Cost-Effectiveness of the State-Based Coverage Criteria All Transitions, All Round-Trip Paths, All Transition Pairs, Paths of Length 2, Paths of Length 3 and Paths of Length 3?

This chapter concerned cost-effectiveness of the state-based coverage criteria all transitions (AT), all round-trip paths (RTP), all transition pairs (ATP), paths of length 2 (LN2), paths of length 3 (LN3) and paths of length 4 (LN4) when applied to a detailed state-based test model using oracle O1.

Results showed that LN2 provided the smallest test suite (27 test cases); whereas ATP generated the largest test suite (1,425 test cases). The median among the test-suite sizes was 232 test cases.

In compliance with the test suite sizes, LN2 had the lowest mean values for preparation time (126 seconds). The highest value, 28,819 seconds, was observed for ATP. Criteria LN3 (509 seconds) and RTP (531 seconds) had quite similar values. A large increase from LN3 and RTP was seen for AT (3,995 seconds). The second highest measure collected for preparation time was LN4 with 5,295 seconds.

Results show for execution time that, again, LN2 provided the lowest value (18 seconds). LN3 was measured to 136 seconds, followed by RTP with 489 seconds. Almost doubling the time seen for RTP, LN4 was measured to use 850 seconds on executing the test suite. The second highest time was measured for AT – 2,455 seconds. Finally, execution of the ATP test suite took 3,341 seconds.

Considering the mutation score results ranked by mean from low to high, results showed that LN2 performed significantly poorer than the other coverage criteria (5/15 killed mutants). A large gap was to be found between LN2 and the next result; LN3 killed 14/15 of the mutants. Quite similar, AT resulted in a high mutation score mean, 0.997. The best mutation score mean came with RTP, ATP, and LN4 – all mutants were killed.

The paired Wilcoxon signed-rank test was applied to the replicated data, i.e., for AT, RTP, and ATP. All tests executed on preparation and execution time resulted in significantly

different results – ATP spent significantly more time on both preparation and execution of the test suites than AT and RTP. No significant differences were found in data collected on mutation score.

Conclusions: The results indicate that LN2 might be too weak as a testing strategy. The other testing strategies performed similar with respect to mutation score, but with varying costs – ATP was the most expensive criterion. Having rather similar cost-effectiveness, LN3 and RPT were suggested by the results as the most cost-effective strategies.

14.2 Case Study 2 – How does Varying the Oracle Affect the Cost-Effectiveness?

Case study 2 addressed research question 2 regarding how two different oracles influenced the cost-effectiveness of state-based testing when using the strategies all transitions (AT), all round-trip paths (RTP), all transition pairs (ATP), paths of length 2 (LN2), paths of length 3 (LN3) and paths of length 4 (LN4) on a detailed state-based test model.

Overall, varying the oracle shows that there were small differences in the preparation time. Three of six criteria spent similar or slightly more time when using oracle O2, whereas the remaining three criteria spent slightly less time when using oracle O2. Execution time, on the other hand, was significantly lower when applying oracle O2. Large cost savings was achieved. However, the higher mutation score was negatively affected. For all strategies, oracle O2 obtained lower mutation score means than oracle O1. The greatest difference was found in results for LN2 (149.8 percent difference), followed by ATP (36.9 percent difference). Rather similar differences were seen among LN3 (27.2 percent), AT (25.3 percent), RTP (25 percent), and LN4 (25 percent).

For both oracles, AT, RTP and LN4 performed best in terms of mutation score. Of those three strategies, RTP was the least expensive followed by LN4 and then AT. ATP was as effective as AT, RTP, and LN4, but only when applying O1. ATP in combination with O2 achieved a slightly lower mutation score. Note, however, that ATP was the most expensive strategy. Although a slightly lower mutation score was obtained for O1, LN3 also achieved a good cost-effectiveness – in particular when looking at the cost. Finally, LN2 not only appeared to be the least expensive but also the weakest strategy from a fault-detection perspective.

As we have seen, the combinations of coverage criterion and oracle significantly impact the cost and fault-detection effectiveness of the testing strategies in different directions, in particular the execution cost. We found that the most cost-effective strategy in this study was RTP combined with oracle O1. Note, however, that LN3 combined with O1 obtained cost-effectiveness comparable to RTP.

Conclusions: Minor differences in preparation time were observed when applying oracle O2. Execution time, on the other hand, was significantly lower when applying oracle O2 for all six strategies. The large cost savings when using O2, however, had a negative impact on the effectiveness.

14.3 Case Study 3 – What is the Influence of the Test Model Abstraction Level on the Cost-Effectiveness?

This chapter addressed whether or not varying the level of details in test models affect the cost-effectiveness of the state-based coverage criteria all transitions (AT), all round-trip paths (RTP), all transition pairs (ATP), paths of length 2 (LN2), paths of length 3 (LN3) and paths of length 4 (LN4). Both oracles that were applied in Case Study 2 were also applied in this study.

Results showed that the test-suite sizes significantly decreased when abstracting the test model. However, the number was further reduced by the fact that infeasible test cases were introduced as a result of the sub states being removed in the abstract test model. This implied less control over test data with respect to externally controlled variables and caused infeasible test cases. Only feasible test cases were included in the results.

For oracle O1, the greatest reduction in test-suite size was seen for AT (from 166 to 17 test cases), closely followed by ATP (1425 to 192 test cases). RTP was reduced from 299 to 66 test cases. The LN4 test suite was reduced from 764 to 415 test cases. The remaining strategies were not affected to the same extent; LN2 and LN3 were reduced, by 14.8 and 29.4 percent, respectively. By using oracle O2, the difference in test-suite sizes was overall lower due to lower number of infeasible test cases as compared to using oracle O1, but quite consistent with the results for O1.

Across all criteria, results showed that it takes significantly shorter time for all strategies to prepare test suites from the abstract test model as compared to the complete test model. The greatest difference was seen for ATP (96 percent for both O1 and O2), closely followed by AT (95 percent for O1 and 94 percent for O2). RTP was reduced by 62 percent for O1 and 64 percent for O2. Also LN2, LN3, and LN4 spent less time on generating test suites from the abstract model (from 26 to 40 percent reduction as compared to the complete model).

We saw a similar trend for execution time with great differences between the times spent on executing the test suites generated from the abstract model as compared to the complete model. The largest reduction was achieved by AT (97 percent for O1 and 96 percent for O2), followed by ATP (88 percent for O1 and 80 percent for O2), and RTP (85 percent for O1 and 77 percent for O2). When O1 was combined with the abstract model for LN2, LN3, and LN4, results show that the preparation times were reduced by 11 percent, 38 percent, and 22 percent, respectively. No difference was found between the abstract and complete model when applying O2 to LN2. A minor reduction was found for LN3 (4 percent). The execution time for LN4, on the other hand, was increased by 6 percent.

Significantly smaller test-suite sizes were found for the abstract model versus the complete model for LN2, LN3, and LN4 contra AT, RTP, and ATP. Thus, smaller differences in execution and preparation time would be expected results.

Comparing the two oracles applied to the abstract test model provided rather consistent results as compared to what was seen for the complete test model: Differences in preparation time for the abstract model was in the range from -5 percent to 2 percent compared to -7 percent to 4 percent for the complete model. Regarding execution time, we saw a similar correlation between the abstract and the complete model – for the abstract model, test suites combined with O2 spent from 68 percent to 74 percent less time on execution compared to O1, whereas test suites generated from the complete model combined with O2 spent 72 percent to 83 percent less time on execution compared to O1.

An overall trend in the results for mutation score was that the abstract test models obtained lower mutation score means than what were achieved by the complete test models.

The non-paired Wilcoxon signed-rank test was applied to the replicated data, i.e., for AT, RTP, and ATP. Overall, i.e., considering the three strategies combined with each of O1 and O2, significant differences were found for each of preparation time, execution time and mutation score. Significance was found for $\alpha = 0.01$. Test suites generated from the complete model required both higher preparation and execution time (large effect sizes were found for both measures). On the other hand, the complete models achieved higher mutation scores.

To summarize, the results showed that ATP applied to a detailed model is an expensive strategy. The high fault-detection effectiveness may be at a too high cost. When combined with the abstract model, cost was significantly reduced. The effectiveness was also reduced, but not as much as the cost. On the other extreme, LN2 had the lowest cost but also the lowest effect; at least for the complete model. Results for the abstract model combined with oracle O1 showed similar results to what was found for AT applied to the abstract model. The level of details in the model had an enormous impact on the cost and effectiveness for AT; all mutants were killed by AT when using the complete model, although at a large increase in cost. The complete model combined with oracle O1 obtained as good mutation score as ATP and AT also for RTP and LN4. Of these, RTP had the lowest costs. Overall, the abstract model performs better with oracle O1.

AT, RTP, ATP, and LN4 all provided the highest mutation scores when generated from the complete model used with oracle O1. Of these, RTP had the lowest costs. Using the weaker oracle O2, still based on the complete test model, the effectiveness was slightly reduced: AT, RTP, and LN4 killed 80 % of the mutants. Regarding the abstract test model, we saw that

LN3, LN4, RTP, and ATP killed 87 % of the mutants – the cost of ATP was dramatically reduced as compared to the test suites generated from the complete model.

Conclusions: Reducing the level of detail in the test model significantly influences the cost-effectiveness. Results show that both costs and fault-detection ability are lower for test suites generated from the abstract model as compared to the complete model.

14.4 Case Study 4 – What is the Impact of Sneak-Path Testing on the Cost-Effectiveness?

In Case Study 4, sneak-path testing was applied to the SUT. Recall that 11 of 26 seeded faults were sneak paths.

Complementing state-based testing (Chapter 8–10) with sneak-path testing at an additional cost in preparation and execution time resulted in that the remaining 11 mutants were killed – those 11 mutants were not killed by any of the six state-based coverage criteria. Execution of the sneak-path test suite on the abstract model killed 10/11 sneak paths. This was, however, due to an infeasible test case.

Being equal to the number of states in the SUT, the cost of sneak-path test suites are rather inexpensive as compared to the state-based coverage criteria investigated in this study. Recall that the length of each test case depends on two matters: (1) the length of the path that must be traversed in order to reach the particular state to be tested, and (2) the number of known unexpected events for the state.

The results from this case study demonstrate that both conformance testing and sneak-path testing are complementary in order to detect faults of varying nature. Thus, the results indicate quite strongly that *sneak-path testing is a necessary step in state-based testing due to the following observations: (1) the proportion of sneak paths in the collected fault data was high (42 %), and (2) the presence of sneak paths is undetectable by conformance testing.*

Our results support the recommendation of Binder [31] and the conclusions drawn in the study of Mouchawrab *et al.* [92]: Testing sneak paths is an essential component of state-based testing in practice. The additional cost is justified by the positive influence on fault-detection effectiveness.

Conclusions: The obtained results confirm the importance of including sneak-path testing to improve fault-detection effectiveness of state-based testing, and strongly indicate that sneak-path testing is a necessary step in state-based testing.

14.5 Summary

In summary, this thesis complements and extends existing research on the cost-effectiveness of SBT by:

- developing an extensible and configurable model-based testing tool,
- using an industrial safety-critical control system,
- using real faults collected from a global, industrial field study when evaluating testing strategies,
- comparing six state-based coverage criteria,
- performing a comparison of two test oracles,
- studying the impact of varying the test model abstraction level on cost and effectiveness, and
- studying the benefits of sneak-path testing.

We experienced that the iterative process of generating tests from the specifications was useful in finding inconsistencies not only between the model and the implementation, but also in detecting ambiguities in the specifications. Furthermore, the use of SBT enables predictable testing which is repeatable, and automation in particular enables thorough, systematic testing that may seem impossible to conduct manually. On the other hand, the benefits do not come for free – initial investments not only include training and tool setup; the use of tools also comes with the cost of initial mappings between the test model and the specific programming language, and the cost of test-data generation. Although state-based testing requires an initial investment, changing the test cases for re-test according to changes in the model and code is not as labor intensive as compared to manual testing.

The choice of selecting the type of testing strategy, oracle, and test model abstraction level depends on several factors such as the criticality of the SUT, and the available resources and time. As both cost and effectiveness are reported in this thesis, it is hoped that the presented results will be useful when considering the use of SBT.

Ultimately, Chapter 15 presents suggestions for future work.

15 Future Work

This chapter suggests areas that deserve more focus in future work identified from the gap in existing research and during execution of this study.

15.1 Oracles

When searching for related work, few studies were identified that compare test oracles applied in an SBT-context. A test oracle can check the output of a test in different detailed levels. As we have seen in the few existing studies and from the results of this thesis, the choice of oracle influences both cost and effectiveness. In what way varying the oracle will influence fault detection of the applied test coverage criteria, should be further studied.

15.2 Test Models

There is a lack of research on the interesting aspect of comparing test models of different levels of detail. Several studies focus on reducing the test suite using various reduction techniques. In this thesis, however, we investigate the fault-detection effectiveness of reduced test suites based on a different idea; whereas test-reduction techniques are based on removing tests in a test suite that do not contribute in increasing the fault-detection ability, this thesis rather focuses on abstracting the test model itself by removing details, e.g. removing sub states from composite states. To the author's knowledge, no other studies are conducted on this particular topic in an SBT-context.

15.3 Test Trees

Being non-deterministic approaches, the all transition (AT), all round-trip paths (RTP) and all transition pairs (ATP) criteria may provide a number of different test trees that cover these criteria. Due to the numerous trees, obtained results can differ from tree to tree. Hence, the empirical evaluation of the state-based coverage criteria from this study should be replicated using test tree selection approaches to ensure that the most optimal trees are selected, e.g. like Briand *et al.* [81] which selects the trees with the highest fault-detection ability.

15.4 Model versus Implementation Coverage

Yet another interesting research area is comparisons of model coverage versus implementation coverage to see the influence of differences in coverage on the fault-detection effectiveness.

15.5 Test Data Selection

The selected test data have a large impact on the feasibility of the generated test suites. In this study, the selection of test data for environment variables (i.e., externally controlled variables) was carefully chosen; though only semi-automated, to enable the execution of the test suites. However, the evaluations should be replicated also when varying the test data.

15.6 Cost

Existing research have evaluated state-based coverage criteria. Most studies that regard effectiveness, however, do not report cost. As the attention has mostly been directed towards effectiveness, there is still a lack of empirical results regarding the cost of such testing. Especially, regarding how state-based test criteria perform when being exposed to real faults.

15.7 Cost of Initial Investment

Applying systematic strategies in software testing reduces the dependency between the tester's skills and the achieved test coverage, and furthermore leaves the company with an assurance on how thorough their software is tested. The advantages of state-based testing require an initial investment in tools and training. Future work should explore the cost of said investment.

15.8 Industrial Context

As for any empirical studies, replications of this study are needed. Further evaluations on testing strategies should be carried out, in particular conducted on industrial systems in order to increase the external validity of the results. Future research should aim at evaluating coverage criteria by varying the SUT instead of reusing the same SUT. Moreover, as stated in Chapter 3, existing research shows that extremely few studies[10] apply real faults when using

[10] Table 1 shows two studies where as few as 4/21 and 4/20 faults were real.

mutation analysis for evaluating various testing strategies – the use of artificial faults is prevalent. As a consequence, little is known about how such structured test approaches compares in detecting real faults. Another interesting study would be to compare the results of this study to a study where mutation operators are used to create mutant programs.

15.9 Tools

In general, the community should continuously seek to improve model-based automated tools in order to make model-based testing a feasible approach to the industry. Regarding TRUST in particular, as we saw in Section 5.1.3, more efficient test data generation techniques based on search or optimization techniques [115] should be considered implemented in future versions.

15.10 Guidelines

In order to provide the industry with useful guidelines, we are dependent on future studies in all of the abovementioned areas.

15.11 Choice of Research Method

As a final remark, the case study is a useful research method for software engineering as software engineering takes place within a context. It is important not to factor out the effect of the context when validating technologies for use in industrial development [131]. However, surveys on the use of research methods in software engineering show a fairly low percentage of case studies [131, 132, 64]. More studies should be applied within a realistic context in combination with experiments that offers more control. Experiments provide useful insight, but are, however difficult to execute amongst others due to the large number of variables that cause differences in reality [133] as compared to the few variables that are studied in experiments. Executing several studies following different types of research methods on a particular research area are necessary in order to provide a more “complete” picture.

Bibliography

[1] M. Utting and B. Legeard, Practical Model-Based Testing: A Tools Approach, Morgan-Kaufmann, 2006.

[2] T. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Transactions on Software Engineering,* vol. 4, no. 3, pp. 178-187, 1978.

[3] L. Briand and Y. Labiche, "A UML-Based Approach to System Testing," in *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, 2001.

[4] I. El-Far and J. Whittaker, Model-Based Software Testing, Encyclopedia of Software Engineering (edited by J. J. Marciniak), 2002.

[5] S. Ali, L. Briand, M. Rehman, H. Asghar, M. Iqbal and A. Nadeem, "A State-Based Approach to Integration Testing Based on UML Models," *Information and Software Technology,* vol. 49, pp. 1087-1106, 2007.

[6] A. Neto, R. Subramanyan, M. Vieira and G. Travassos, "A Survey on Model-based Testing Approaches: A Systematic Review," in *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, Atlanta, Georgia, 2007.

[7] D. Drusinsky, Modeling and Verification using UML Statecharts: A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking, 1st edition ed., Newnes, 2006.

[8] M. Vieira, X. Song, G. Matos, S. Storck, R. Tanikella and B. Hasling, "Applying Model-Based Testing to Healthcare Products: Preliminary Experiences," in *Proceedings of the 30th International Conference on Software Engineering*, Leipzig, Germany, 2008.

[9] "D-MINT, Deployment of Model-Based Technologies to Industrial Testing," [Online]. Available: http://www.d-mint.org/. [Accessed 13 June 2012].

[10] J. Feldstein, "Model-based Testing using IBM Rational Functional Tester," *DeveloperWorks, IBM,* 2005.

[11] Y. Gurevich, W. Schulte, N. Tillmann and M. Veanes, "Model-Based Testing with SpecExplorer," Microsoft research, 2009.

[12] L. Briand, M. Di Penta and Y. Labiche, "Assessing and Improving State-Based Class Testing: A Series of Experiments," *IEEE Transactions on SoftWare Engineering,* vol. 30, no. 11, pp. 770-793, 2004.

[13] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[14] N. Holt, B. Anda, K. Asskildt, L. Briand, J. Endresen and S. Frøystein, "Experiences with Precise State Modeling in an Industrial Safety Critical System," in *Critical Systems Development Using Modeling Lanuguages, CSDUML'06*, Genova, 2006.

[15] J. Offutt and A. Abdurazik, "Generating Tests from UML Specifications," in *Proceedings of the 2nd International Conference on the Unified Modeling Language*, 1999.

[16] J. Offutt, S. Liu, A. Abdurazik and P. Ammann, "Generating Test Data from State-based Specifications," *Software: Testing, Verification and Reliability,* vol. 13, no. 1, pp. 25-53, 2003.

[17] "MOTES," [Online]. Available: http://www.elvior.ee/motes. [Accessed 13 June 2012].

[18] A. Hartman and K. Nagin, "The AGEDIS Tools for Model Based Testing," in *International Symposium on Software Testing and Analysis (ISSTA '04)*, 2004.

[19] "Working with TestConductor and Automatic Test Generation (ATG)," IBM, [Online]. Available: http://publib.boulder.ibm.com/infocenter/rhaphlp/v7r6/index.jsp?topic=%2Fcom.ibm.rhp.integ.testingtools.doc%2Ftopics%2Frhp_r_dm_vendor_doc_testing.html. [Accessed 13 June 2012].

[20] D. Seifert, "The TEAGER Tool Suite: Test Execution and Generation Framework for Reactive Systems," [Online]. Available: http://user.cs.tu-berlin.de/~seifert/teager.html. [Accessed September 2009].

[21] "Conformiq Tool Suite," [Online]. Available: http://www.conformiq.com/. [Accessed 13 June 2012].

[22] L. Briand, Y. Labiche and Y. Wang, "Using Simulation to Empirically Investigate Test Coverage Criteria Based on Statechart," in *ICSE '04 Proceedings of the 26th International Conference on Software Engineering*, 2004.

[23] L. Briand, "A Critical Analysis of Empirical Research in Software Testing," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, 2007.

[24] P. Black, V. Okun and Y. Yesha, "Mutation Operators for Specifications," in *ASE'2000, 15th Automated Software Engineering Conference*, Grenoble, France, 2000.

[25] A. Offutt, A. Lee, G. Rothermel, R. Untch and C. Zapf, "An Experimental Determination of Sufficient Mutation Operators," *ACM Trans. Software Eng. and Methodology,* vol. 5, no. 2, pp. 99-118, 1996.

[26] J. Andrews, L. Briand and Y. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?," in *In ICSE '05: Proceedings of the 27th international conference on Software engineering*, 2005.

[27] M. Thévenod-Fosse and P. Daran, "Software Error Analysis: a Real Case Study involving Real Faults and Mutations," in *In Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 1996.

[28] J. Andrews, L. Briand, Y. Labiche and A. Namin, "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria," *IEEE Transactions on Software Engineering,* vol. 32, no. 8, pp. 608-624, August 2006.

[29] M. Heimdahl and D. George, "Test-Suite Reduction for Model Based Tests: Effects on Test Quality and Implications for Testing," in *19th IEEE International Conference on Automated Software Engineering (ASE'04)*, Linz, 2004.

[30] W. Wong, J. Horgan, S. London and A. Mathur, "Effect of Test Set Minimization of Fault Detection Effectiveness," in *International Conference on Software Engineering, Proceedings of the 17th international conference on Software engineering*, Seattle, 1995.

[31] R. Binder, Testing Object-Oriented Systems, Addison-Wesley, 2000.

[32] T. Pender, UML Bible, Wiley, 2003.

[33] S. Ali, H. Hemmati, N. Holt, E. Arisholm and L. Briand, "Technical Report 2010-01: Model Transformations as a Strategy to Automate Model-Based Testing: A Tool and Industrial Case Studies," Simula Research Laboratory, Lysaker, 2010.

[34] N. Holt, E. Arisholm and L. Briand, "Technical Report 2009-06: An Eclipse

Plug-in for the Flattening of Concurrency and Hierarchy in UML State Machines," Simula Research Laboratory, Lysaker, 2009.

[35] J. McGregor and D. Sykes, A Practical Guide to Testing Object-Oriented Software, Addison-Wesley, 2001.

[36] M. Utting, A. Pretschner and B. Legeard, "A Taxonomy of Model-Based Testing," Working paper series. University of Waikato, Department of Computer Science. No. 04/2006, 2006.

[37] A. Gupta and P. Jalote, "An Approach for Experimentally Evaluating Effectiveness and Efficiency of Coverage Criteria for Software Testing," *International Journal of Software Tools and Technology Transfer,* vol. 10, no. 2, pp. 145-160, 2008.

[38] J. Spivey, The Z Notation: A Reference Manual, 2nd ed., Prentice-Hall: Englewood Cliffs, N.J., 1992.

[39] J. Abrial, The B-BOOK: Assigning Programs to Meanings, Cambridge University Press: Cambridge, 1996.

[40] "The Java Modeling Language (JML)," [Online]. Available: http://www.eecs.ucf.edu/~leavens/JML/. [Accessed 13 June 2012].

[41] "Smartesting," [Online]. Available: http://www.smartesting.com/index.php/cms/en/home. [Accessed 27 July 2010].

[42] D. Harel and A. Naamad, "The STATEMATE Semantics of Statecharts," *ACM Transactions on Software Engineering and Methodology,* vol. 5, no. 4, p. 293–333, 1996.

[43] R. Miles and K. Hamilton, Learning UML 2.0, First edition ed., O'Reilly, 2006.

[44] "Unified Modelling Language Specification, Version 2.0," Object Management Group, September 2009. [Online]. Available: http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML.

[45] L. Lavagno, G. Martin and B. Selic, UML for Real: Design of Embedded Real-Time Systems, Springer, 2003.

[46] T. Weigert and R. Reed, "Specifying Telecommunications Systems with UML," in *UML for Real: Design of Embedded Real-time Systems*, Kluwer Academic Publishers, 2003, pp. 301-322.

[47] S. Sauer and G. Engels, "UML-Based Behavior Specification of Interactive Multimedia Applications," in *Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01)*, 2001.

[48] "Papyrus," September 2009. [Online]. Available: http://www.papyrusuml.org.

[49] "Rational Software Architect for WebSphere Software," IBM, [Online]. Available: http://www-01.ibm.com/software/awdtools/swarchitect/websphere/. [Accessed 14 June 2012].

[50] J. Zhang, C. Xu and X. Wang, "Path-Oriented Test Data Generation Using Symbolic Execution and Constraint Solving Techniques," in *Second International Conference on Software Engineering and Formal Methods (SEFM'04)*, 2004.

[51] R. Lefticaru and F. Ipate, "Automatic State-Based Test Generation Using Genetic Algorithms," in *9th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2008.

[52] D. Chiorean, M. Bortes, D. Corutiu, C. Botiza and A. Cârcu, "OCLE," [Online]. Available: http://lci.cs.ubbcluj.ro/ocle/. [Accessed 14 June 2012].

[53] C. Hein, T. Ritter and M. Wagner, "Open Source Library for OCL," [Online]. Available: http://oslo-project.berlios.de/. [Accessed 14 June 2012].

[54] "IBM OCL Parser," [Online]. Available: http://www-01.ibm.com/software/awdtools/library/standards/ocl-download.html. [Accessed 14 June 2012].

[55] M. Egea, "EyeOCL Software," [Online]. Available: http://www.bm1software.com/eos/. [Accessed 14 June 2012].

[56] A. Offutt, Y. Xiong and S. Liu, "Criteria for Generating Specification-Based Tests," in *In Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, 1999.

[57] Y.-S. Ma, J. Offutt and Y. Kwon, "MuJava : An Automated Class Mutation System," *Software: Testing, Verification and Reliability,* vol. 15, no. 2, pp. 97-133, 2005.

[58] I. Sommerville, Software Engineering, Addison-Wesley, 2001.

[59] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell and A. Wesslén,

Experimentation in Software Engineering: An Introduction, Kluwer Academic Publishers, 2000.

[60] V. Basili, "The Role of Experimentation in Software Engineering: Past, Current, and Future," in *Proceedings of ICSE-18*, 1996.

[61] D. Sjoberg, T. Dybå and M. Jørgensen, "The Future of Empirical Methods in Software Engineering Research," in *FOSE '07 2007 Future of Software Engineering*, Washington, DC, 2007.

[62] S. Sørumgård, Verification of Process Conformance in Empirical Studies of Software Development, PhD Thesis, Department of Computer and Information Sciences, The Norwegian University of Science and Technology, Norway, 1997.

[63] W. Shadish, T. Cook and D. Campbell, Experimental an Quasi-Experimental Designs for Generalized Causal Inference, Houghton Mifflin Company, 2002.

[64] R. Yin, Case Study Research Design and Methods, Sage Publications, 2003.

[65] W. J. Dzidek, Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance, Faculty of Mathematics and Natural Sciences, University of Oslo, 2008.

[66] R. Stake, The Art of Case Study Research, SAGE Publications, 1995.

[67] E. Babbie, Survey Research Methods, Wadsworth, 1990.

[68] G. Booch, J. Rumbaugh and I. Jacobson, The Unified Modeling Language User Guide, Addison Wesley, 1999.

[69] D. Lee and M. Yannakakis, "Principles and Methods of Testing Finite State Machines - A Survey," *Proceedings of the IEEE,* vol. 84, no. 8, pp. 1090 - 1123, August 1996.

[70] A. Abdurazik, P. Ammann, W. Ding and J. Offutt, "Evaluation of Three Specification-based Testing Criteria," in *Proceedings of the 6th IEEE International Conference on Complex Computer Systems*, 2000.

[71] P. Ammann, P. Black and W. Majurski, "Using Model Checking to Generate Tests from Specifications," in *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, 1998.

[72] A. P. Mathur, "On the Relative Strengths of Data Flow and Mutation Based Test Adequacy Criteria," in *Proceedings of the Sixth Annual Pacific Northwest*

Software Quality Conference, 1991.

[73] H. Hong, Y. Kim, S. Cha, D. Bae and H. Ural, "A Test Sequence Selection Method for Statecharts," *Software: Testing, Verification and Reliability,* vol. 10, no. 4, pp. 203-227, 2000.

[74] H. Ural, "Test sequence selection based on static data flow analysis," *Computer Communications,* vol. 10, no. 5, pp. 234-242, 1987.

[75] H. Ural and B. Yang, "A Test Sequence Selection Method for Protocol Testing," *IEEE Transactions on Communications,* vol. 39, no. 4, pp. 514-523, 1991.

[76] H. Ural and A. Williams, "Test Generation by Exposing Control and Data Dependencies within System Specifications in SDL," in *In Proceedings of IFIP 6th International Conference on Formal Description Techniques, FORTE'93*, 1993.

[77] K. Bogdanov and M. Holcombe, "Statechart Testing Method for Aircraft Control Systems," *Software Testing, Verification and Reliability,* vol. 11, no. 1, pp. 39-54, 2001.

[78] P. Chevalley and P. Thévenod-Fosse, "An Empirical Evaluation of Statistical Testing Designed from UML State Diagrams: the Flight Guidance System Case Study," in *Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE'01)*, 2001.

[79] P. Chevalley and P. Thévenod-Fosse, "Automated Generation of Statistical Test Cases from UML State Diagrams," in *Computer Software and Applications Conference, 2001. COMPSAC 2001*, 2001.

[80] G. Antoniol, L. Briand, M. Di Penta and Y. Labiche, "A Case Study Using the Round-Trip Strategy for State-Based Class Testing," in *Proceedings of the 13 th International Symposium on Software Reliability Engineering (ISSRE'02)*, 2002.

[81] L. Briand, Y. Labiche and Q. Lin, "Improving Statechart Testing Criteria Using Data Flow Information," in *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, 2005.

[82] A. Paradkar, "Plannable Test Selection Criteria for FSMs Extracted from Operational Specifications," in *Proc. of Int. Symp. on Software Reliability Eng. '2004*, 2004.

[83] A. Paradkar, "Case Studies on Fault Detection Effectiveness of Model Based

Test Generation Techniques," in *Advances in Model-Based Software Testing (A-MOST'05)*, 2005.

[84] B. Legeard, F. Peureux and M. Utting, "A Comparison of the BTT and TTF Test-Generation Methods," in *Proc. of ZB '02: Formal Specification and Development in Z and B*, 2002.

[85] S. Mouchawrab, L. Briand and Y. Labiche, "Assessing, Comparing, and Combining Statechart-based testing and Structural testing: An Experiment," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, 2007.

[86] M. Staats, M. Whalen and M. Heimdahl, "Better Testing Through Oracle Selection," in *ICSE '11: Proceeding of the 33rd International Conference on Software Engineering*, 2011.

[87] M. Staats, G. Gay and M. Heimdahl, "Automated Oracle Creation Support, or: How I Learned to Stop Worrying About Fault Propagation and Love Mutation Testing," in *ICSE 2012*, 2012.

[88] M. Staats, M. Whalen and M. Heimdahl, "Programs, Tests, and Oracles: The Foundations of Testing Revisited," in *ICSE '11: Proceeding of the 33rd International Conference on Software Engineering*, 2011.

[89] L. Briand, W. Dzidek and Y. Labiche, "Using Aspect-Oriented Programming to Instrument OCL Contracts in Java," Carleton University, 2004.

[90] A. Memon, I. Banerjee and A. Nagarajan, "What Test Oracle Should I Use for Effective GUI Testing?," in *Proceedings of the IEEE International Conference on Automated Software Engineering*, Montreal, Canada, 2003.

[91] Q. Xie and A. Memon, "Designing and Comparing Automated Test Oracles for GUI-Based Software Applications," *ACM Transactions on Software Engineering and Methodology,* vol. 16, no. 1, February 2007.

[92] S. Mouchawrab, L. Briand, Y. Labiche and M. Di Penta, "Assessing, Comparing, and Combining State Machine-Based Testing and Structural Testing: A Series of Experiments," *IEEE Transactions on Software Engineering,* vol. 37, no. 2, pp. 161 - 187, 2011.

[93] "Model-Based Testing Tools," [Online]. Available: http://en.wikipedia.org/wiki/Model-based_testing_tools. [Accessed 13 June

2012].

[94] S. Weißleder, "Partition Test Generator (ParTeG)," [Online]. Available: http://parteg.sourceforge.net/. [Accessed June13 2012].

[95] "MARTE: Modeling and Analysis of Real-time and Embedded systems," [Online]. Available: http://www.omgmarte.org/node/4. [Accessed 13 June 2012].

[96] R. Cavarra, C. Crichton, J. Davies, A. Hartman and L. Mounier, "Using UML for Automatic Test Generation," in *In International Symposium on Software Testing and Analysis (ISSTA '02)*, 2002.

[97] "Poseidon for UML," [Online]. Available: http://www.gentleware.com. [Accessed 13 June 2012].

[98] L. Briand, Y. Labiche and Q. Lin, "Improving the Coverage Criteria of UML State Machines using Data Flow Analysis," *Software Testing, Verification and Reliability,* vol. 20, no. 3, p. 177–207, 2010.

[99] M. Khalil and Y. Labiche, "On the Round Trip Path Testing Strategy," in *IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)*, 2010.

[100] H. Hemmati and L. Briand, "An Industrial Investigation of Similarity Measures for Model-Based Test Case Selection," in *IEEE, Software Reliability Engineering (ISSRE)*, 2010.

[101] H. Hemmati, A. Arcuri and L. Briand, "Empirical Investigation of the Effects of Test Suite Properties on Similarity-Based Test Case Selection," in *IEEE, Software Testing, Verification and Validation (ICST)*, 2011.

[102] E. Cartaxo, P. Machado and F. Neto, "On the Use of a Similarity Function for Test Case Selection in the Context of Model-Based Testing," *Software Testing, Verification and Reliability,* vol. 21, no. 2, p. 75–100, 2011.

[103] H. Gomaa, Designing Concurrent, Distributed, and Real-Time Applications with UML, Addison-Wesley, 2000.

[104] A. Kleppe, J. Warmer and W. Bast, MDA Explained: The Model Driven Architecture: Practice and Promise, Addison Wesley, 2003.

[105] "UML Profile for Modeling QoS and Fault Tolerance Characteristics and Mechanisms," [Online]. Available: http://www.omg.org/spec/QFTP/1.1/.

[Accessed 13 June 2012].

[106] S. Kansomkeat and W. Rivepiboon, "Automated-generating test case using UML statechart diagrams," in *Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, 2003.

[107] T. Chen, F.-C. Kuo, R. Merkel and T. Tse, "Adaptive Random Testing: The ART of Test Case Diversity," *Journal of Systems and Software,* vol. 83, no. 1, p. 60–66, 2010.

[108] T. Jussila, J. Dubrovin, T. Junttila, T. Latvala and I. Porres, "Model Checking Dynamic and Hierarchical UML State Machines," in *Proceedings of the 3rd Workshop on Model Design and Validation (MoDeVa06)*, 2006.

[109] "Kermeta – Breathe Life into Your Metamodels," [Online]. Available: http://www.kermeta.org/. [Accessed 13 June 2012].

[110] P. Huber, "The Model Transformation Language Jungle - An Evaluation and Extension of Existing Approaches," Business Informatics Group: Institut für Softwaretechnik und Interaktive Systeme, 2008.

[111] L. Briand, Y. Labiche and Y. Wang, "Using Simulation to Empirically Investigate Test Coverage Criteria on Statecharts," Carleton University Technical Report SCE-02-09, 2002.

[112] "MOFScript Home page," [Online]. Available: http://www.eclipse.org/gmt/mofscript/. [Accessed 14 June 2012].

[113] "OMG's MetaObject Facility," [Online]. Available: http://www.omg.org/mof/. [Accessed 14 June 2012].

[114] R. DeMillo and A. Offutt, "Constraint-Based Automatic Test Data Generation," *IEEE Transactions on Software Engineering,* vol. 17, no. 9, pp. 900-910, 1991.

[115] P. McMinn, "Search-Based Software Test Data Generation: A Survey," *Software: Testing, Verification, and Reliability,* vol. 14, no. 2, pp. 105-156, 2004.

[116] H. Mössenböck, M. Löberbauer and A. Wöß, "The Compiler Generator Coco/R," [Online]. Available: http://ssw.jku.at/coco/. [Accessed 14 June 2012].

[117] T. Parr, "ANTLR v3," [Online]. Available: http://www.antlr.org/. [Accessed 14 June 2012].

[118] S. Liang, Java Native Interface: Programmer's Guide and Specification, Addison-Wesley Publishing, 1999.

[119] G. McCluskey, "Using Java Reflection," [Online]. Available: http://java.sun.com/developer/technicalArticles/ALT/Reflection/. [Accessed 14 June 2012].

[120] N. Denzin and Y. Lincoln, The Sage Handbook of Qualitative Research, Third edition ed., Sage Publications, 2005.

[121] B. Kitchenham, "Evaluating Software Engineering Methods and Tool," *Software Engineering Notes,* vol. 22, no. 4, pp. 21-24, July 1997.

[122] P. Runeson and M. Höst, "Guidelines for Conducting and Reporting Case Study Research in Software Engineering," *Empirical Software Engineering,* vol. 14, no. 2, pp. 131-164, December 2009.

[123] L. Briand and Y. Labiche, "Empirical Studies of Software Testing Techniques: Challenges, Practical Strategies, and Future Research," *ACM SIGSOFT Software Engineering Notes,* vol. 29, no. 5, pp. 1-3, 2004.

[124] "OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2.," [Online]. Available: http://www.omg.org/spec/UML/2.1.2/. [Accessed 14 June 2012].

[125] M. Alshraideh and L. Bottaci, "Search-Based Software Test Data Generation for String Data Using Program-Specific Search Operators," *Software Testing, Verification and Reliability,* vol. 16, no. 3, p. 175–203, 2006.

[126] "JMP," [Online]. Available: http://www.jmp.com/. [Accessed 15 July 2010].

[127] "The R Project for Statistical Computing," [Online]. Available: http://www.r-project.org/. [Accessed 14 June 2012].

[128] W. Conover, Practical Nonparametric Statistics, Wiley, 1999.

[129] M. Schervish, Theory of Statistics, Springer, 1995.

[130] S. Kim, J. Clark and J. McDermid, "The Rigorous Generation of Java Mutation Using Hazop," in *12 th International Conference Software & Systems Engineering and their Applications (ICSSEA`99)*, 1999.

[131] J. Segal, A. Grinyer and H. Sharp, "The Type of Evidence Produced by Empirical Software Engineers," in *Proceedings of the Workshop on Realising Evidence-*

Based Software Engineering, ICSE 2005, 2005.

[132] R. Glass, I. Vessey and V. Ramesh, "Research in Software Engineering: An Analysis of the Literature," *Information and Software Technology,* vol. 44, no. 8, pp. 491-506, 2002.

[133] V. Basili, "What's So Hard About Replication of Software Engineering Experiments?," [Online]. Available: http://www.cs.umd.edu/~basili/presentations/RESER%20Keynote.pdf. [Accessed 14 June 2012].

A. Overview of Research Activities

Period	ID	Activity	Role
2006 June – 2007 Feb	A1.	Specification, design, and implementation of a module in a safety-critical software system in ABB using UML state machines and the extended state design pattern.	Participatory role, in collaboration with ABB and supervisors.
2006 Sept	A2.	Presentation of accepted work shop paper, CSDUML '06.	Primary role, in collaboration with supervisors and ABB.
2007 Jan – 2007 May	A3.	Manual state-based testing of the module.	Primary role.
2007 May – 2007 Nov	A4.	Development of the same system in accordance to typical development ABB development (baseline).	Primary role, in collaboration with external developer Simen Hagen.
2008 Jan – 2008 April	A5.	Write-up of report from the action research related to activity A1, A3, and A4.	Primary role.
	A6.	Field study on maintenance. The following sub tasks were conducted:	
2007 Jan – 2007 Feb		A6.1) A guideline for how to use the extended state design pattern.	Primary role.
2007 March – 2007 April		A6.2) A pilot study was designed and conducted. The subject was a researcher/developer from ABB.	Participatory role, in collaboration with main supervisor.
2008 May – 2008 Sept		A6.3) Design of a maintenance field study in ABB.	Primary role, in collaboration with main supervisor. Kai Hansen assisted recruitment of subjects.
2008 Oct – 2008 Nov		A6.4) Execution of field study in ABB's departments in Västerås, Baden and Shanghai.	Primary role.

2008 Dec		A6.5) Analysis of the collected data.	Primary role.
2009 Jan – 2009 April	A7.	Development of an automated test tool in cooperation with the PhD students Shaukat Ali and Hadi Hemmati. I was responsible for the following tasks:	Participatory role, in collaboration with Hadi Hemmati, Shaukat Ali and supervisors.
2009 Jan – 2009 May		A7.1) Development of a transformation tool, an Eclipse plug-in, for flattening complex UML state machines.	Primary role, advised by supervisors.
2009 May – 2009 June		A7.2) Adjustment of TRUST to support C++ to enable the generation of concrete test cases from abstract test cases. The tool was also extended to support the all round-trip path transition coverage. The test cases were generated from the state machines developed in activity A1 and used to test the state-based implementation of ALC.	Primary role.
2009 June – 2009 Sept	A8.	Technical report based on the tool development.	Participatory role, in collaboration with Shaukat Ali, Hadi Hemmati, and supervisors.
2009 Oct – 2010 Feb	A9.	Technical report based on the state-machine flattening Eclipse plug-in.	Primary role.
2009 July – 2010 May	A10.	Evaluation of testing strategies, including the work of extending TRUST to support generation of test cases that satisfy additional five coverage criteria and an additional oracle. The testing strategies were evaluated by analyzing the fault-detection effectiveness of the test suites using real fault-data (collected in activity A6) and mutation analysis.	Primary role. Extensions are based on the tool TRUST developed in collaboration with Hadi Hemmati and Shaukat Ali.
2010 Jan – 2010 July	A11.	Write-up of the thesis. Full-time position as a PhD student.	Primary role.
2010 Aug – 2012 June	A12.	Continuing write-up of the thesis. Part-time position (10-20 percent) as a PhD student.	Primary role.
2012 May	A13.	Conference paper based on Case Study 4 is submitted for publication.	Primary role.
2012 April – 2012 Aug	A14.	Journal paper based on Case Study 1–4 is submitted for publication.	Primary role.

B. Semi-Structured Literature Review

JOURNALS

Database	Publication	Search string	Where	All Papers	Included papers
ACM	TOSEM	“state based testing”	title	0	0
ACM	TOSEM	“state-based testing”	title	0	0
ACM	TOSEM	“state machine testing”	title	0	0
ACM	TOSEM	“state-machine testing”	title	0	0
ACM	TOSEM	“model based testing”	title	0	0
ACM	TOSEM	“model-based testing”	title	0	0
ACM	TOSEM	“state based testing”	abstract	1	0
ACM	TOSEM	“state-based testing”	abstract	1	0
ACM	TOSEM	“state machine testing”	abstract	0	0
ACM	TOSEM	“state-machine testing”	abstract	0	0
ACM	TOSEM	“model based testing”	abstract	0	0
ACM	TOSEM	“model-based testing”	abstract	0	0
# unique papers				**1**	**0**
Database	**Publication**	**Search string**	**Where**	**All Papers**	**Included papers**
IEEE	TSE	“state based testing”	title	0	0
IEEE	TSE	“state-based testing”	title	0	0
IEEE	TSE	“state machine testing”	title	0	0
IEEE	TSE	“state-machine testing”	title	0	0
IEEE	TSE	“model based testing”	title	0	0
IEEE	TSE	“model-based testing”	title	0	0
IEEE	TSE	“state based testing”	abstract	0	0
IEEE	TSE	“state-based testing”	abstract	0	0
IEEE	TSE	“state machine testing”	abstract	1	1
IEEE	TSE	“state-machine testing”	abstract	1	1
IEEE	TSE	“model based testing”	abstract	1	1
IEEE	TSE	“model-based testing”	abstract	1	1
# unique papers				**1**	**1**
Database	**Publication**	**Search string**	**Where**	**All Papers**	**Included papers**
Wiley	STVR	“state based testing”	title	0	0
Wiley	STVR	“state-based testing”	title	0	0
Wiley	STVR	“state machine testing”	title	1	0
Wiley	STVR	“state-machine testing”	title	1	0
Wiley	STVR	“model based testing”	title	4	1
Wiley	STVR	“model-based testing”	title	4	1
Wiley	STVR	“state based testing”	abstract	3	0
Wiley	STVR	“state-based testing”	abstract	3	0
Wiley	STVR	“state machine testing”	abstract	1	1
Wiley	STVR	“state-machine testing”	abstract	1	1
Wiley	STVR	“model based testing”	abstract	9	1
Wiley	STVR	“model-based testing”	abstract	9	1

# unique papers	12	2
Total # unique papers from TOSEM, TSE, and STVR	14	3

CONFERENCES

Database	Publication	Search string	Where	All Papers	Included papers
ACM	ISSTA	“state based testing”	title	0	0
ACM	ISSTA	“state-based testing”	title	0	0
ACM	ISSTA	“state machine testing”	title	0	0
ACM	ISSTA	“state-machine testing”	title	0	0
ACM	ISSTA	“model based testing”	title	0	0
ACM	ISSTA	“model-based testing”	title	0	0
ACM	ISSTA	“state based testing”	abstract	0	0
ACM	ISSTA	“state-based testing”	abstract	0	0
ACM	ISSTA	“state machine testing”	abstract	0	0
ACM	ISSTA	“state-machine testing”	abstract	0	0
ACM	ISSTA	“model based testing”	abstract	0	0
ACM	ISSTA	“model-based testing”	abstract	0	0
# unique papers				**0**	**0**
Database	**Publication**	**Search string**	**Where**	**All Papers**	**Included papers**
IEEE	ICST	“state based testing”	title	0	0
IEEE	ICST	“state-based testing”	title	0	0
IEEE	ICST	“state machine testing”	title	0	0
IEEE	ICST	“state-machine testing”	title	0	0
IEEE	ICST	“model based testing”	title	3	0
IEEE	ICST	“model-based testing”	title	3	0
IEEE	ICST	“state based testing”	abstract	0	0
IEEE	ICST	“state-based testing”	abstract	0	0
IEEE	ICST	“state machine testing”	abstract	0	0
IEEE	ICST	“state-machine testing”	abstract	0	0
IEEE	ICST	“model based testing”	abstract	7	1
IEEE	ICST	“model-based testing”	abstract	7	1
# unique papers				**8**	**1**
Database	**Publication**	**Search string**	**Where**	**All Papers**	**Included papers**
IEEE	ISSRE	“state based testing”	title	0	0
IEEE	ISSRE	“state-based testing”	title	0	0
IEEE	ISSRE	“state machine testing”	title	0	0
IEEE	ISSRE	“state-machine testing”	title	0	0
IEEE	ISSRE	“model based testing”	title	0	0
IEEE	ISSRE	“model-based testing”	title	0	0
IEEE	ISSRE	“state based testing”	abstract	1	1
IEEE	ISSRE	“state-based testing”	abstract	1	1
IEEE	ISSRE	“state machine testing”	abstract	1	1
IEEE	ISSRE	“state-machine testing”	abstract	1	1
IEEE	ISSRE	“model based testing”	abstract	0	0

IEEE	ISSRE	“model-based testing”	abstract	0	0
# unique papers				**2**	**2**
Total # unique papers from ISSTA, ICST, and ISSRE				**10**	**3**

C. State Machines – Original Version

Mode State Machine

The diagram can be provided upon request.

Drive Enable State Machine

The diagram can be provided upon request.

D. State Machines – Modified Version of SUT

Mode State Machine

The diagram can be provided upon request.

Drive Enable State Machine

The diagram can be provided upon request.

E. State Machine – Experiment Version (Complete Test Model)

The diagram displays the Mode state machine which was involved in the change task. State `ExtraSlow` and its belonging transitions are the only differences from the state machine shown in Appendix D.

The diagram can be provided upon request.

F. State Machine – Experiment Version (Abstract Test Model)

The diagram can be provided upon request.

G. Data Material – AT, RTP, and ATP – Complete Model – Oracle O1

All transitions (AT), complete test model, and state-invariant oracle (O1)

AT Complete O1		*GEN. TREE*	*GEN. TEST SUITE*	*BUILD TEST SUITE*	PREPARE TEST SUITE	EXECUTE TEST SUITE	MUTATION SCORE
Test Suite	**Test-suite size**	*Sec.*	*Sec.*	*Sec.*	*Sec.*	*Sec.*	
AT1	166	*1858*	*399*	*249*	**2506**	**1881**	**1**
AT2	165	*2034*	*402*	*240*	**2676**	**3102**	**1**
AT3	169	*2172*	*441*	*338*	**2951**	**2115**	**1**
AT4	166	*1858*	*389*	*292*	**2539**	**2315**	**1**
AT5	168	*2236*	*543*	*204*	**2983**	**2625**	**1**
AT6	168	*2236*	*473*	*267*	**2976**	**2432**	**1**
AT7	162	*2346*	*425*	*292*	**3063**	**2383**	**1**
AT8	167	*2445*	*490*	*368*	**3303**	**2810**	**1**
AT9	170	*2567*	*488*	*257*	**3312**	**2458**	**1**
AT10	162	*2346*	*422*	*245*	**3013**	**1858**	**1**
AT11	168	*2236*	*409*	*211*	**2856**	**2432**	**0.933**
AT12	173	*2861*	*408*	*270*	**3539**	**3361**	**1**
AT13	164	*2936*	*476*	*354*	**3766**	**2851**	**1**
AT14	165	*2034*	*470*	*258*	**2762**	**3122**	**1**
AT15	162	*2346*	*487*	*275*	**3108**	**2163**	**1**
AT16	166	*1858*	*479*	*261*	**2598**	**1854**	**1**
AT17	171	*3269*	*464*	*241*	**3974**	**2169**	**1**
AT18	163	*15124*	*378*	*279*	**15781**	**2088**	**1**
AT19	162	*2346*	*413*	*269*	**3028**	**2020**	**1**
AT20	169	*2172*	*430*	*270*	**2872**	**2298**	**1**
AT21	160	*3675*	*361*	*246*	**4282**	**1765**	**1**
AT22	160	*3675*	*443*	*367*	**4485**	**2210**	**1**
AT23	163	*15124*	*477*	*338*	**15939**	**2022**	**1**
AT24	164	*2936*	*492*	*230*	**3658**	**3096**	**1**
AT25	168	*2236*	*506*	*206*	**2948**	**3617**	**1**
AT26	165	*2034*	*443*	*249*	**2726**	**2274**	**1**
AT27	168	*2236*	*434*	*342*	**3012**	**2370**	**1**
AT28	165	*2034*	*467*	*262*	**2763**	**2776**	**1**
AT29	173	*2861*	*536*	*288*	**3685**	**2564**	**1**
AT30	165	*2034*	*470*	*242*	**2746**	**2631**	**1**
Avg	**165.9**	*3270.8*	*450.5*	*273.7*	**3995**	**2455.4**	**0.998**

All round-trip paths (RTP), complete test model, and state-invariant oracle (O1)

RTP Complete O1		*GEN. TREE*	*GEN. TEST SUITE*	*BUILD TEST SUITE*	**PREPARE TEST SUITE**	**EXECUTE TEST SUITE**	**MUTATION SCORE**
Test Suite	**Test-suite size**	*Sec.*	*Sec.*	*Sec.*	*Sec.*	*Sec.*	
RTP1	328	*105*	*128*	*374*	**607**	**465**	**1**
RTP2	328	*66*	*108*	*406*	**580**	**419**	**1**
RTP3	328	*85*	*99*	*408*	**592**	**341**	**1**
RTP4	328	*116*	*83*	*353*	**552**	**524**	**1**
RTP5	328	*71*	*96*	*354*	**521**	**434**	**1**
RTP6	328	*68*	*82*	*367*	**517**	**409**	**1**
RTP7	328	*68*	*83*	*359*	**510**	**517**	**1**
RTP8	328	*89*	*84*	*349*	**522**	**497**	**1**
RTP9	328	*67*	*85*	*334*	**486**	**528**	**1**
RTP10	328	*66*	*85*	*333*	**484**	**552**	**1**
RTP11	328	*98*	*83*	*348*	**529**	**515**	**1**
RTP12	328	*109*	*82*	*343*	**534**	**514**	**1**
RTP13	328	*106*	*85*	*349*	**540**	**536**	**1**
RTP14	328	*114*	*91*	*356*	**561**	**525**	**1**
RTP15	328	*84*	*90*	*338*	**512**	**607**	**1**
RTP16	328	*93*	*87*	*340*	**520**	**511**	**1**
RTP17	328	*92*	*81*	*354*	**527**	**515**	**1**
RTP18	328	*77*	*87*	*346*	**510**	**406**	**1**
RTP19	328	*95*	*87*	*376*	**558**	**493**	**1**
RTP20	328	*81*	*89*	*360*	**530**	**483**	**1**
RTP21	328	*97*	*84*	*354*	**535**	**511**	**1**
RTP22	328	*84*	*84*	*344*	**512**	**468**	**1**
RTP23	328	*76*	*86*	*347*	**509**	**495**	**1**
RTP24	328	*84*	*89*	*334*	**507**	**533**	**1**
RTP25	328	*76*	*87*	*353*	**516**	**528**	**1**
RTP26	328	*78*	*107*	*374*	**559**	**460**	**1**
RTP27	328	*77*	*101*	*365*	**543**	**410**	**1**
RTP28	328	*70*	*86*	*377*	**533**	**460**	**1**
RTP29	328	*74*	*92*	*354*	**520**	**514**	**1**
RTP30	328	*76*	*81*	*355*	**512**	**493**	**1**
Avg	**328**	*84.7*	*89.7*	*356.8*	**531.3**	**488.8**	**1**

All transition-pairs (ATP), complete test model, and state-invariant oracle (O1)

ATP Complete O1		*GEN. TREE*	*GEN. TEST SUITE*	*BUILD TEST SUITE*	**PREPARE TEST SUITE**	**EXECUTE TEST SUITE**	**MUTATION SCORE**
Test Suite	**Test-suite size**	*Sec.*	*Sec.*	*Sec.*	**Sec.**	**Sec.**	
ATP1	1425	*26340*	*520*	*1722*	**28582**	**3377**	**1**
ATP2	1425	*26340*	*429*	*1794*	**28563**	**3260**	**1**
ATP3	1425	*26340*	*474*	*1685*	**28499**	**2680**	**1**
ATP4	1425	*26340*	*488*	*1850*	**28678**	**3465**	**1**
ATP5	1425	*26340*	*527*	*2260*	**29127**	**3418**	**1**
ATP6	1425	*26340*	*507*	*2374*	**29221**	**3582**	**1**
ATP7	1425	*26340*	*483*	*1916*	**28739**	**3747**	**1**
ATP8	1425	*26340*	*531*	*2258*	**29129**	**2665**	**1**
ATP9	1425	*26340*	*469*	*1991*	**28800**	**2612**	**1**
ATP10	1425	*26340*	*518*	*1860*	**28718**	**2607**	**1**
ATP11	1425	*26340*	*629*	*2131*	**29100**	**3110**	**1**
ATP12	1425	*26340*	*512*	*1720*	**28572**	**2861**	**1**
ATP13	1425	*26340*	*386*	*1851*	**28577**	**2958**	**1**
ATP14	1425	*26340*	*518*	*2018*	**28876**	**3618**	**1**
ATP15	1425	*26340*	*549*	*1632*	**28521**	**3944**	**1**
ATP16	1425	*26340*	*636*	*2361*	**29337**	**2830**	**1**
ATP17	1425	*26340*	*519*	*2539*	**29398**	**3124**	**1**
ATP18	1425	*26340*	*555*	*2045*	**28940**	**3310**	**1**
ATP19	1425	*26340*	*528*	*1827*	**28695**	**2976**	**1**
ATP20	1425	*26340*	*576*	*1859*	**28775**	**3542**	**1**
ATP21	1425	*26340*	*542*	*1597*	**28479**	**3959**	**1**
ATP22	1425	*26340*	*523*	*1553*	**28416**	**3175**	**1**
ATP23	1425	*26340*	*491*	*1546*	**28377**	**3909**	**1**
ATP24	1425	*26340*	*493*	*2038*	**28871**	**3978**	**1**
ATP25	1425	*26340*	*506*	*2158*	**29004**	**3906**	**1**
ATP26	1425	*26340*	*529*	*1987*	**28856**	**3298**	**1**
ATP27	1425	*26340*	*548*	*2012*	**28900**	**3774**	**1**
ATP28	1425	*26340*	*563*	*1890*	**28793**	**3643**	**1**
ATP29	1425	*26340*	*573*	*1973*	**28886**	**3385**	**1**
ATP30	1425	*26340*	*520*	*2274*	**29134**	**3527**	**1**
Avg	**1425**	*26340.0*	*521.4*	*1957.4*	**28818.8**	**3341.3**	**1**

H. Data Material – AT, RTP, and ATP – Complete Model – Oracle O2

All transitions (AT), complete test model, and state-pointer oracle (O2)

AT Complete O2		*GEN. TREE*	*GEN. TEST SUITE*	*BUILD TEST SUITE*	PREPARE TEST SUITE	EXECUTE TEST SUITE	MUTATION SCORE
Test Suite	Test-suite size	*Sec.*	*Sec.*	*Sec.*	*Sec.*	*Sec.*	
AT1	166	*1858*	*206*	*185*	**2249**	**293**	**0.8**
AT2	165	*2034*	*212*	*197*	**2443**	**386**	**0.8**
AT3	169	*2172*	*223*	*197*	**2592**	**325**	**0.8**
AT4	166	*1858*	*210*	*235*	**2303**	**571**	**0.8**
AT5	168	*2236*	*261*	*192*	**2689**	**318**	**0.8**
AT6	168	*2236*	*250*	*329*	**2815**	**336**	**0.8**
AT7	162	*2346*	*245*	*182*	**2773**	**495**	**0.8**
AT8	167	*2445*	*262*	*190*	**2897**	**454**	**0.8**
AT9	170	*2567*	*253*	*197*	**3017**	**365**	**0.8**
AT10	162	*2346*	*220*	*193*	**2759**	**373**	**0.8**
AT11	168	*2236*	*219*	*186*	**2641**	**312**	**0.7**
AT12	173	*2861*	*262*	*274*	**3397**	**427**	**0.7**
AT13	164	*2936*	*276*	*210*	**3422**	**407**	**0.8**
AT14	165	*2034*	*262*	*185*	**2481**	**510**	**0.8**
AT15	162	*2346*	*241*	*186*	**2773**	**442**	**0.8**
AT16	166	*1858*	*240*	*179*	**2277**	**411**	**0.8**
AT17	171	*3269*	*259*	*209*	**3737**	**359**	**0.8**
AT18	163	*15124*	*240*	*259*	**15623**	**425**	**0.8**
AT19	162	*2346*	*258*	*196*	**2800**	**410**	**0.8**
AT20	169	*2172*	*211*	*208*	**2591**	**348**	**0.8**
AT21	160	*3675*	*227*	*199*	**4101**	**475**	**0.8**
AT22	160	*3675*	*212*	*220*	**4107**	**354**	**0.8**
AT23	163	*15124*	*241*	*298*	**15663**	**433**	**0.8**
AT24	164	*2936*	*272*	*185*	**3393**	**487**	**0.8**
AT25	168	*2236*	*212*	*183*	**2631**	**434**	**0.8**
AT26	165	*2034*	*225*	*185*	**2444**	**386**	**0.8**
AT27	168	*2236*	*230*	*183*	**2649**	**423**	**0.8**
AT28	165	*2034*	*241*	*179*	**2454**	**556**	**0.8**
AT29	173	*2861*	*238*	*201*	**3300**	**472**	**0.8**
AT30	165	*2034*	*226*	*176*	**2436**	**477**	**0.8**
Avg	**165.9**	***3270.8***	***237.8***	***206.6***	**3715.2**	**415.5**	**0.796**

All round-trip paths (RTP), complete test model, and state-pointer oracle (O2)

RTP Complete O2		*GEN. TREE*	*GEN. TEST SUITE*	*BUILD TEST SUITE*	**PREPARE TEST SUITE**	**EXECUTE TEST SUITE**	**MUTATION SCORE**
Test Suite	**Test-suite size**	*Sec.*	*Sec.*	*Sec.*	*Sec.*	*Sec.*	
RTP1	328	*105*	*105*	*341*	**551**	**99**	**0.8**
RTP2	328	*66*	*66*	*344*	**476**	**102**	**0.8**
RTP3	328	*85*	*85*	*347*	**517**	**84**	**0.8**
RTP4	328	*116*	*116*	*443*	**675**	**119**	**0.8**
RTP5	328	*71*	*71*	*336*	**478**	**89**	**0.8**
RTP6	328	*68*	*68*	*342*	**478**	**89**	**0.8**
RTP7	328	*68*	*68*	*350*	**486**	**86**	**0.8**
RTP8	328	*89*	*89*	*348*	**526**	**85**	**0.8**
RTP9	328	*67*	*67*	*360*	**494**	**87**	**0.8**
RTP10	328	*66*	*66*	*348*	**480**	**80**	**0.8**
RTP11	328	*98*	*98*	*402*	**598**	**101**	**0.8**
RTP12	328	*109*	*109*	*370*	**588**	**98**	**0.8**
RTP13	328	*106*	*106*	*348*	**560**	**97**	**0.8**
RTP14	328	*114*	*114*	*381*	**609**	**91**	**0.8**
RTP15	328	*84*	*84*	*377*	**545**	**87**	**0.8**
RTP16	328	*93*	*93*	*363*	**549**	**95**	**0.8**
RTP17	328	*92*	*92*	*374*	**558**	**88**	**0.8**
RTP18	328	*77*	*77*	*369*	**523**	**89**	**0.8**
RTP19	328	*95*	*95*	*454*	**644**	**104**	**0.8**
RTP20	328	*81*	*81*	*390*	**552**	**91**	**0.8**
RTP21	328	*97*	*97*	*379*	**573**	**87**	**0.8**
RTP22	328	*84*	*84*	*355*	**523**	**106**	**0.8**
RTP23	328	*76*	*76*	*347*	**499**	**105**	**0.8**
RTP24	328	*84*	*84*	*363*	**531**	**107**	**0.8**
RTP25	328	*76*	*76*	*345*	**497**	**103**	**0.8**
RTP26	328	*78*	*78*	*352*	**508**	**101**	**0.8**
RTP27	328	*77*	*77*	*343*	**497**	**98**	**0.8**
RTP28	328	*70*	*70*	*344*	**484**	**97**	**0.8**
RTP29	328	*74*	*74*	*332*	**480**	**100**	**0.8**
RTP30	328	*76*	*76*	*345*	**497**	**96**	**0.8**
Avg		***84.7***	***84.7***	***363.1***	**532.5**	**95.4**	**0.8**

All transition-pairs (ATP), complete test model, and state-pointer oracle (O2)

ATP Complete O2		*GEN. TREE*	*GEN. TEST SUITE*	*BUILD TEST SUITE*	**PREPARE TEST SUITE**	**EXECUTE TEST SUITE**	**MUTATION SCORE**
Test Suite	**Test-suite size**	*Sec.*	*Sec.*	*Sec.*	*Sec.*	*Sec.*	
AT1	1425	*26340*	*420*	*1746*	**28506**	**536**	**0.80**
AT2	1425	*26340*	*383*	*1829*	**28552**	**525**	**0.73**
AT3	1425	*26340*	*430*	*2319*	**29089**	**773**	**0.73**
AT4	1425	*26340*	*417*	*1595*	**28352**	**679**	**0.73**
AT5	1425	*26340*	*451*	*1514*	**28305**	**563**	**0.73**
AT6	1425	*26340*	*428*	*1544*	**28312**	**504**	**0.73**
AT7	1425	*26340*	*449*	*1654*	**28443**	**594**	**0.73**
AT8	1425	*26340*	*437*	*1560*	**28337**	**489**	**0.73**
AT9	1425	*26340*	*437*	*1556*	**28333**	**577**	**0.73**
AT10	1425	*26340*	*408*	*1673*	**28421**	**558**	**0.73**
AT11	1425	*26340*	*637*	*1713*	**28690**	**608**	**0.73**
AT12	1425	*26340*	*447*	*1901*	**28688**	**649**	**0.73**
AT13	1425	*26340*	*426*	*1711*	**28477**	**508**	**0.80**
AT14	1425	*26340*	*562*	*2646*	**29548**	**723**	**0.80**
AT15	1425	*26340*	*424*	*2530*	**29294**	**623**	**0.67**
AT16	1425	*26340*	*434*	*2304*	**29078**	**717**	**0.73**
AT17	1425	*26340*	*514*	*2662*	**29516**	**611**	**0.73**
AT18	1425	*26340*	*430*	*1723*	**28493**	**712**	**0.73**
AT19	1425	*26340*	*401*	*1873*	**28614**	**496**	**0.73**
AT20	1425	*26340*	*370*	*1791*	**28501**	**745**	**0.73**
AT21	1425	*26340*	*650*	*1803*	**28793**	**668**	**0.73**
AT22	1425	*26340*	*464*	*1845*	**28649**	**509**	**0.73**
AT23	1425	*26340*	*465*	*2033*	**28838**	**446**	**0.60**
AT24	1425	*26340*	*429*	*1621*	**28390**	**429**	**0.73**
AT25	1425	*26340*	*454*	*1630*	**28424**	**494**	**0.73**
AT26	1425	*26340*	*491*	*1636*	**28467**	**462**	**0.73**
AT27	1425	*26340*	*382*	*1650*	**28372**	**464**	**0.73**
AT28	1425	*26340*	*450*	*1995*	**28785**	**502**	**0.73**
AT29	1425	*26340*	*486*	*1716*	**28542**	**463**	**0.73**
AT30	1425	*26340*	*509*	*1566*	**28415**	**441**	**0.73**
Avg		*26340.0*	*456.2*	*1844.6*	**28640.8**	**568.9**	**0.733**

I. Data Material – AT, RTP, and ATP – Abstract Model – Oracle O1

All transitions (AT), abstract test model, and state-invariant oracle (O1)

AT Abstract O1		*GEN. TREE*	*GEN. TEST SUITE*	*BUILD TEST SUITE*	**PREPARE TEST SUITE**	**EXECUTE TEST SUITE**	**MUTATION SCORE**
Test Suite	**Test-suite size**	*Sec.*	*Sec.*	*Sec.*	*Sec.*	*Sec.*	
AT1	33	*110*	*28*	*47*	**185**	**44**	**0.20**
AT2	33	*110*	*22*	*45*	**177**	**79**	**0.20**
AT3	31	*119*	*26*	*44*	**189**	**91**	**0.20**
AT4	34	*123*	*28*	*46*	**197**	**53**	**0**
AT5	33	*110*	*30*	*46*	**186**	**57**	**0.20**
AT6	35	*139*	*27*	*50*	**216**	**129**	**0.53**
AT7	35	*139*	*26*	*45*	**210**	**40**	**0.2**
AT8	33	*110*	*22*	*46*	**178**	**51**	**0.2**
AT9	33	*110*	*24*	*44*	**178**	**105**	**0.67**
AT10	31	*119*	*25*	*44*	**188**	**70**	**0.20**
AT11	33	*110*	*23*	*47*	**180**	**102**	**0.80**
AT12	33	*110*	*22*	*45*	**177**	**115**	**0.60**
AT13	32	*149*	*27*	*70*	**246**	**84**	**0.20**
AT14	34	*123*	*24*	*45*	**192**	**39**	**0.20**
AT15	34	*123*	*21*	*47*	**191**	**91**	**0.40**
AT16	33	*110*	*19*	*44*	**173**	**53**	**0.20**
AT17	32	*149*	*25*	*52*	**226**	**61**	**0.20**
AT18	31	*119*	*22*	*56*	**197**	**35**	**0.20**
AT19	37	*899*	*23*	*50*	**972**	**71**	**0.20**
AT20	32	*149*	*21*	*62*	**232**	**38**	**0.20**
AT21	33	*110*	*23*	*44*	**177**	**35**	**0.20**
AT22	33	*110*	*24*	*47*	**181**	**61**	**0.27**
AT23	32	*149*	*23*	*51*	**223**	**42**	**0.20**
AT24	31	*119*	*25*	*50*	**194**	**37**	**0.20**
AT25	31	*119*	*23*	*55*	**197**	**29**	**0.20**
AT26	29	*128*	*21*	*49*	**198**	**52**	**0.20**
AT27	35	*139*	*19*	*48*	**206**	**57**	**0.20**
AT28	34	*123*	*22*	*45*	**190**	**51**	**0.20**
AT29	33	*110*	*23*	*45*	**178**	**99**	**0.20**
AT30	29	*128*	*26*	*57*	**211**	**61**	**0.20**
Avg	**32.7**	*148.8*	*23.8*	*48.9*	**221.5**	**64.4**	**0.262**

All round-trip paths (RTP), abstract test model, and state-invariant oracle (O1)

RTP Abstract O1		*GEN. TREE*	*GEN. TEST SUITE*	*BUILD TEST SUITE*	PREPARE TEST SUITE	EXECUTE TEST SUITE	MUTATION SCORE
Test Suite	**Test-suite size**	*Sec. Gen. Tree*	*Sec. Gen. Test Suite*	*Sec. Build*	**Sec.**	**Sec. Run**	
RTP1	89	*89*	*19*	*87*	**195**	**75**	**0.87**
RTP2	89	*89*	*20*	*87*	**196**	**72**	**0.87**
RTP3	89	*89*	*23*	*78*	**190**	**64**	**0.87**
RTP4	89	*89*	*15*	*88*	**192**	**62**	**0.87**
RTP5	89	*89*	*18*	*87*	**194**	**62**	**0.87**
RTP6	89	*89*	*16*	*102*	**207**	**62**	**0.87**
RTP7	89	*89*	*17*	*115*	**221**	**52**	**0.87**
RTP8	89	*89*	*24*	*78*	**191**	**58**	**0.87**
RTP9	89	*89*	*32*	*78*	**199**	**60**	**0.87**
RTP10	89	*89*	*20*	*83*	**192**	**77**	**0.87**
RTP11	89	*89*	*15*	*82*	**186**	**60**	**0.87**
RTP12	89	*89*	*15*	*143*	**247**	**52**	**0.87**
RTP13	89	*89*	*28*	*102*	**219**	**75**	**0.87**
RTP14	89	*89*	*21*	*161*	**271**	**71**	**0.87**
RTP15	89	*89*	*22*	*198*	**309**	**88**	**0.87**
RTP16	89	*89*	*21*	*121*	**231**	**84**	**0.87**
RTP17	89	*89*	*16*	*93*	**198**	**84**	**0.87**
RTP18	89	*89*	*21*	*95*	**205**	**89**	**0.87**
RTP19	89	*89*	*18*	*134*	**241**	**100**	**0.87**
RTP20	89	*89*	*24*	*90*	**203**	**73**	**0.87**
RTP21	89	*89*	*15*	*82*	**186**	**76**	**0.87**
RTP22	89	*89*	*15*	*78*	**182**	**74**	**0.87**
RTP23	89	*89*	*17*	*76*	**182**	**74**	**0.87**
RTP24	89	*89*	*17*	*78*	**184**	**76**	**0.87**
RTP25	89	*89*	*17*	*77*	**183**	**75**	**0.87**
RTP26	89	*89*	*16*	*81*	**186**	**73**	**0.87**
RTP27	89	*89*	*16*	*77*	**182**	**73**	**0.87**
RTP28	89	*89*	*16*	*77*	**182**	**75**	**0.87**
RTP29	89	*89*	*15*	*78*	**182**	**73**	**0.87**
RTP30	89	*89*	*15*	*79*	**183**	**75**	**0.87**
Avg	**89**	*89.0*	*18.8*	*96.2*	**204.0**	**72.1**	**0.87**

All transition pairs (ATP), abstract test model, and state-invariant oracle (O1)

ATP Abstract O1		*GEN. TREE*	*GEN. TEST SUITE*	*BUILD TEST SUITE*	**PREPARE TEST SUITE**	**EXECUTE TEST SUITE**	**MUTATION SCORE**
Test Suite	**Test-suite size**	*Sec.*	*Sec.*	*Sec.*	*Sec.*	*Sec.*	
ATP1	301	*703*	*104*	*312*	**1119**	**313**	**0.867**
ATP2	301	*703*	*78*	*317*	**1098**	**312**	**0.867**
ATP3	301	*703*	*80*	*319*	**1102**	**358**	**0.867**
ATP4	301	*703*	*89*	*319*	**1111**	**478**	**0.867**
ATP5	301	*703*	*92*	*315*	**1110**	**313**	**0.867**
ATP6	301	*703*	*88*	*319*	**1110**	**406**	**0.867**
ATP7	301	*703*	*90*	*316*	**1109**	**215**	**0.867**
ATP8	301	*703*	*89*	*344*	**1136**	**366**	**0.867**
ATP9	301	*703*	*91*	*344*	**1138**	**338**	**0.867**
ATP10	301	*703*	*87*	*334*	**1124**	**427**	**0.867**
ATP11	301	*703*	*72*	*314*	**1089**	**401**	**0.867**
ATP12	301	*703*	*83*	*353*	**1139**	**348**	**0.867**
ATP13	301	*703*	*80*	*315*	**1098**	**343**	**0.867**
ATP14	301	*703*	*83*	*324*	**1110**	**415**	**0.867**
ATP15	301	*703*	*82*	*348*	**1133**	**467**	**0.867**
ATP16	301	*703*	*85*	*313*	**1101**	**495**	**0.867**
ATP17	301	*703*	*84*	*321*	**1108**	**330**	**0.867**
ATP18	301	*703*	*86*	*339*	**1128**	**441**	**0.867**
ATP19	301	*703*	*87*	*331*	**1121**	**509**	**0.867**
ATP20	301	*703*	*89*	*330*	**1122**	**356**	**0.867**
ATP21	301	*703*	*93*	*354*	**1150**	**405**	**0.867**
ATP22	301	*703*	*83*	*333*	**1119**	**504**	**0.867**
ATP23	301	*703*	*85*	*317*	**1105**	**515**	**0.867**
ATP24	301	*703*	*86*	*323*	**1112**	**394**	**0.867**
ATP25	301	*703*	*89*	*314*	**1106**	**436**	**0.867**
ATP26	301	*703*	*89*	*325*	**1117**	**296**	**0.867**
ATP27	301	*703*	*89*	*331*	**1123**	**372**	**0.867**
ATP28	301	*703*	*85*	*316*	**1104**	**362**	**0.867**
ATP29	301	*703*	*88*	*310*	**1101**	**403**	**0.867**
ATP30	301	*703*	*92*	*311*	**1106**	**528**	**0.867**
Avg	**301**	*703.0*	*86.6*	*325.4*	**1115.0**	**394.9**	**0.867**

J. Data Material – AT, RTP, and ATP – Abstract Model – Oracle O2

All transitions (AT), abstract test model, and state-pointer oracle (O2)

AT Abstract O2		*GEN. TREE*	*GEN. TEST SUITE*	*BUILD TEST SUITE*	**PREPARE TEST SUITE**	**EXECUTE TEST SUITE**	**MUTATION SCORE**
Test Suite	**Test-suite size**	*Sec.*	*Sec.*	*Sec.*	*Sec.*	*Sec.*	
AT1	33	*110*	*22*	*62*	**194**	**24**	**0.20**
AT2	33	*110*	*14*	*50*	**174**	**18**	**0**
AT3	31	*119*	*16*	*44*	**179**	**19**	**0**
AT4	34	*123*	*15*	*50*	**188**	**18**	**0**
AT5	33	*110*	*14*	*45*	169	**16**	**0.33**
AT6	35	*139*	*15*	*49*	**203**	**21**	**0**
AT7	35	*139*	*14*	*46*	**199**	**11**	**0**
AT8	33	*110*	*15*	*49*	**174**	**14**	**0.40**
AT9	33	*110*	*14*	*49*	**173**	**14**	**0**
AT10	31	*119*	*13*	*53*	185	**13**	**0.53**
AT11	33	*110*	*19*	*81*	**210**	**25**	**0.40**
AT12	33	*110*	*15*	*56*	**181**	**36**	**0**
AT13	32	*149*	*17*	*57*	**223**	**13**	**0**
AT14	34	*123*	*15*	*63*	**201**	**12**	**0.20**
AT15	34	*123*	*14*	*57*	194	**18**	**0**
AT16	33	*110*	*22*	*44*	**176**	**10**	**0**
AT17	32	*149*	*16*	*43*	**208**	**15**	**0**
AT18	31	*119*	*14*	*42*	**175**	**12**	**0**
AT19	37	*899*	*14*	*52*	**965**	**20**	**0**
AT20	32	*149*	*14*	*43*	206	**11**	**0**
AT21	33	*110*	*14*	*47*	**171**	**13**	**0**
AT22	33	*110*	*14*	*44*	**168**	**18**	**0**
AT23	32	*149*	*14*	*43*	**206**	**13**	**0**
AT24	31	*119*	*14*	*42*	**175**	**9**	**0**
AT25	31	*119*	*14*	*44*	177	**11**	**0**
AT26	29	*128*	*14*	*43*	**185**	**19**	**0**
AT27	35	*139*	*14*	*46*	**199**	**22**	**0**
AT28	34	*123*	*14*	*46*	**183**	**9**	**0.20**
AT29	33	*110*	*14*	*44*	**168**	**37**	**0**
AT30	29	*128*	*15*	*42*	185	**16**	**0.20**
Avg	**32.7**	*148.8*	*15.1*	*49.2*	**213.1**	**16.9**	**0.076**

All round-trip paths (RTP), abstract test model, and state-pointer oracle (O2)

RTP Abstract O2		*GEN. TREE*	*GEN. TEST SUITE*	*BUILD TEST SUITE*	**PREPARE TEST SUITE**	**EXECUTE TEST SUITE**	**MUTATION SCORE**
Test Suite	**Test-suite size**	*Sec.*	*Sec.*	*Sec.*	*Sec.*	*Sec.*	
RTP1	89	*89*	*19*	*79*	**187**	**45**	**0.6**
RTP2	89	*89*	*25*	*92*	**206**	**23**	**0.6**
RTP3	89	*89*	*14*	*97*	**200**	**20**	**0.6**
RTP4	89	*89*	*15*	*80*	**184**	**21**	**0.6**
RTP5	89	*89*	*15*	*83*	**187**	**22**	**0.6**
RTP6	89	*89*	*15*	*81*	**185**	**21**	**0.6**
RTP7	89	*89*	*12*	*85*	**186**	**17**	**0.6**
RTP8	89	*89*	*13*	*80*	**182**	**19**	**0.6**
RTP9	89	*89*	*14*	*86*	**189**	**17**	**0.6**
RTP10	89	*89*	*13*	*86*	**188**	**23**	**0.6**
RTP11	89	*89*	*13*	*88*	**190**	**27**	**0.6**
RTP12	89	*89*	*17*	*95*	**201**	**15**	**0.6**
RTP13	89	*89*	*14*	*78*	**181**	**17**	**0.6**
RTP14	89	*89*	*15*	*76*	**180**	**20**	**0.6**
RTP15	89	*89*	*17*	*78*	**184**	**21**	**0.6**
RTP16	89	*89*	*13*	*82*	**184**	**14**	**0.6**
RTP17	89	*89*	*13*	*78*	**180**	**19**	**0.6**
RTP18	89	*89*	*13*	*77*	**179**	**15**	**0.6**
RTP19	89	*89*	*13*	*106*	**208**	**15**	**0.6**
RTP20	89	*89*	*13*	*90*	**192**	**21**	**0.6**
RTP21	89	*89*	*35*	*122*	**246**	**47**	**0.6**
RTP22	89	*89*	*17*	*144*	**250**	**22**	**0.6**
RTP23	89	*89*	*22*	*91*	**202**	**21**	**0.6**
RTP24	89	*89*	*16*	*89*	**194**	**25**	**0.6**
RTP25	89	*89*	*18*	*91*	**198**	**23**	**0.6**
RTP26	89	*89*	*17*	*91*	**197**	**22**	**0.6**
RTP27	89	*89*	*16*	*98*	**203**	**22**	**0.6**
RTP28	89	*89*	*14*	*82*	**185**	**26**	**0.6**
RTP29	89	*89*	*16*	*82*	**187**	**20**	**0.6**
RTP30	89	*89*	*20*	*87*	**196**	**22**	**0.6**
Avg	**89**	*89.0*	*16.2*	*89.1*	**194.4**	**22.1**	**0.6**

All transition pairs (ATP), abstract test model, and state-pointer oracle (O2)

ATP Abstract O2		*GEN. TREE*	*GEN. TEST SUITE*	*BUILD TEST SUITE*	**PREPARE TEST SUITE**	**EXECUTE TEST SUITE**	**MUTATION SCORE**
Test Suite	**Test-suite size**	*Sec.*	*Sec.*	*Sec.*	*Sec.*	*Sec.*	
ATP1	301	*703*	*93*	*355*	**1151**	**93**	**0.60**
ATP2	301	*703*	*71*	*318*	**1092**	**83**	**0.60**
ATP3	301	*703*	*71*	*314*	**1088**	**99**	**0.60**
ATP4	301	*703*	*75*	*312*	**1090**	**117**	**0.67**
ATP5	301	*703*	*74*	*313*	**1090**	**91**	**0.67**
ATP6	301	*703*	*82*	*318*	**1103**	**122**	**0.67**
ATP7	301	*703*	*77*	*314*	**1094**	**65**	**0.60**
ATP8	301	*703*	*76*	*325*	**1104**	**116**	**0.67**
ATP9	301	*703*	*77*	*318*	**1098**	**106**	**0.60**
ATP10	301	*703*	*82*	*337*	**1122**	**124**	**0.60**
ATP11	301	*703*	*69*	*316*	**1088**	**115**	**0.60**
ATP12	301	*703*	*63*	*348*	**1114**	**122**	**0.67**
ATP13	301	*703*	*69*	*324*	**1096**	**100**	**0.67**
ATP14	301	*703*	*70*	*344*	**1117**	**105**	**0.60**
ATP15	301	*703*	*82*	*326*	**1111**	**135**	**0.60**
ATP16	301	*703*	*69*	*325*	**1097**	**101**	**0.67**
ATP17	301	*703*	*69*	*333*	**1105**	**126**	**0.67**
ATP18	301	*703*	*68*	*388*	**1159**	**132**	**0.67**
ATP19	301	*703*	*68*	*349*	**1120**	**177**	**0.67**
ATP20	301	*703*	*95*	*341*	**1139**	**113**	**0.60**
ATP21	301	*703*	*65*	*472*	**1240**	**122**	**0.60**
ATP22	301	*703*	*80*	*409*	**1192**	**130**	**0.60**
ATP23	301	*703*	*93*	*372*	**1168**	**145**	**0.67**
ATP24	301	*703*	*92*	*315*	**1110**	**97**	**0.67**
ATP25	301	*703*	*100*	*322*	**1125**	**123**	**0.60**
ATP26	301	*703*	*114*	*394*	**1211**	**95**	**0.67**
ATP27	301	*703*	*89*	*447*	**1239**	**124**	**0.60**
ATP28	301	*703*	*126*	*415*	**1244**	**117**	**0.60**
ATP29	301	*703*	*115*	*550*	**1368**	**142**	**0.60**
ATP30	301	*703*	*71*	*385*	**1159**	**140**	**0.67**
Avg	**301**	*703.0*	*81.5*	*356.6*	**1141.1**	**115.9**	**0.63**

K. Statistical Tests for Case Study 1

The paired Wilcoxon signed-rank test was used for comparing the testing strategies. The following abbreviates are used in the tests:

- AT: the all transitions coverage criterion
- RTP: the all round-trip paths coverage criterion
- ATP: the all transition pairs coverage criterion
- Complete: the detailed test model
- Abstract: the test model where contents in super states were removed
- O1: the state-invariant oracle
- O2: the state-pointer oracle
- Prep: the time spent on preparing the test suite
- Exec: the time spent on executing the test suite
- Mut. score: the mutation score; that is, the number of non-equivalent mutants killed divided by the total number of non-equivalent mutants

The following paired Wilcoxon signed-rank test was run:

```
wilcox.test(x, y, paired=TRUE, conf.level=0.99)
a.statistic.default(x$strategy.measure.test model.oracle,
x$strategy.measure.test model.oracle)
```

Preparation time

RTP versus ATP

```
H_O: Complete_O1_RTP_Prep.time = Complete_O1_ATP_Prep.time
data:  x$RTP.prep.complete.O1 and x$ATP.prep.complete.O1
V = 0, p-value = 1.863e-09
A statistic = 0
```

AT versus RTP

```
H_0: Complete_O1_AT_Prep.time = Complete_RTP_O1_Prep.time
```

```
data:  x$AT.prep.complete.O1 and x$RTP.prep.complete.O1
V = 465, p-value = 1.863e-09
A statistic = 1
```

AT versus ATP

```
H_0: Complete_O1_AT_Prep.time = Complete_ATP_O1_Prep.time
data:  x$AT.prep.complete.O1 and x$ATP.prep.complete.O1
V = 0, p-value = 1.863e-09 (sign!)
A statistics = 1
```

Execution time

RPT versus ATP

```
H_O: Complete_O1_RTP_Exec.time = Complete_ATP_O1_Exec.time
data:  x$RTP.exec.complete.O1 and x$ATP.exec.complete.O1
V = 0, p-value = 1.863e-09
A statistic = 0
```

AT versus RTP

```
H_0: Complete_O1_AT_Exec.time = Complete_RTP_O1_Exec.time
data:  x$AT.exec.complete.O1 and x$RTP.exec.complete.O1
V = 465, p-value = 1.824e-06
A statistics = 1
```

AT versus ATP

```
H_0: Complete_O1_AT_Exec.time = Complete_ATP_O1_Exec.time
data:  x$AT.exec.complete.O1 and x$ATP.exec.complete.O1
V = 9, p-value = 4.5e-06
A statistic = 0.08555556
```

Mutation score

RTP versus ATP

```
H_O: Complete_O1_RTP_Mut.score = Complete_ATP_O1_Mut.score
data:  x$RTP.Mut.Score.O1 and x$ATP.Mut.Score.O1
V = 0, p-value = NA
A statistics = 0.5
```

AT versus RTP

```
H_0: Complete_O1_AT_Mut.score = Complete_RTP_O1_Mut.score
data:  x$AT.Mut.Score.O1 and x$RTP.Mut.Score.O1
V = 0, p-value = NA
A statistics = 0.5
```

AT versus ATP

```
H_0: Complete_O1_AT_Mut.score = Complete_ATP_O1_Mut.score
data:  x$AT.Mut.Score.O1 and x$ATP.Mut.Score.O1
V = 0, p-value = NA
A statistics = 0.5 (no effect)
```

L. Statistical Tests for Case Study 2

The paired Wilcoxon signed-rank test was used for comparing the test oracles. The following abbreviates are used in the tests:

- AT: the all transitions coverage criterion
- RTP: the all round-trip paths coverage criterion
- ATP: the all transition pairs coverage criterion
- Complete: the detailed test model
- Abstract: the test model where contents in super states were removed
- O1: the state-invariant oracle
- O2: the state-pointer oracle
- Prep: the time spent on preparing the test suite
- Exec: the time spent on executing the test suite
- Mut. score: the mutation score; that is, the number of non-equivalent mutants killed divided by the total number of non-equivalent mutants

The following paired Wilcoxon signed-rank test was run:

```
wilcox.test(x, y, paired=TRUE, conf.level=0.99)
a.statistic.default(x$strategy.measure.test model.oracle,
x$strategy.measure.test model.oracle)
```

Preparation time

AT – The state-invariant oracle versus the state-pointer oracle

```
AT.prep.complete.[O1|O2]: 1.824e-06 (they are different)
a.statistic.default(sample1 = f$AT.prep.complete.O1, sample2 =
f$AT.prep.complete.O2)
A statistic = 0.6722222
95% confidence interval for A = [0.521, 0.794]
99% confidence interval for A = [0.474, 0.824]
```

RTP – The state-invariant oracle versus the state-pointer oracle

```
RTP.-"-: p=1 (no SSD)
```

ATP – The state-invariant oracle versus the state-pointer oracle

```
ATP.-"-: p=0.03272 (they are different)
a.statistic.default(sample1 = f$ATP.prep.complete.O1, sample2 =
f$ATP.prep.complete.O2)
A statistic = 0.7227778
95% confidence interval for A = [0.573, 0.835]
99% confidence interval for A = [0.524, 0.860]
```

Execution time

AT – The state-invariant oracle versus the state-pointer oracle

```
AT.exec.complete.[O1|O2]: 1.824e-06
a.statistic.default(sample1 = f$AT.exec.complete.O1, sample2 =
f$AT.exec.complete.O2)
A statistic = 1
95% confidence interval for A = [NA, NA]
99% confidence interval for A = [NA, NA]
```

RTP – The state-invariant oracle versus the state-pointer oracle

```
RTP.-"-: 1.822e-06
a.statistic.default(sample1 = f$RTP.exec.complete.O1, sample2 =
f$RTP.exec.complete.O2)
A statistic = 1
95% confidence interval for A = [NA, NA]
99% confidence interval for A = [NA, NA]
```

ATP – The state-invariant oracle versus the state-pointer oracle

```
ATP.-"-: 1.863e-09
```

```
a.statistic.default(sample1 = f$ATP.exec.complete.O1, sample2 =
f$ATP.exec.complete.O2)
A statistic = 1
95% confidence interval for A = [NA, NA]
99% confidence interval for A = [NA, NA]
```

Mutations score – The complete test model

AT – The state-invariant oracle versus the state-pointer oracle

```
AT.mut.score[O1|O2]: p=1.08e-07
a.statistic.default(sample1 = f$AT.Mut.Score.O1, sample2 =
f$AT.Mut.Score.O2)
A statistic = 1
95% confidence interval for A = [NA, NA]
99% confidence interval for A = [NA, NA]
```

RTP – The state-invariant oracle versus the state-pointer oracle

```
RTP.mut.score[O1|O2]: p=4.616e-08
a.statistic.default(sample1 = f$RTP.Mut.Score.O1, sample2 =
f$RTP.Mut.Score.O2)
A statistic = 1
95% confidence interval for A = [NA, NA]
99% confidence interval for A = [NA, NA]
```

ATP – The state-invariant oracle versus the state-pointer oracle

```
ATP.mut.score[O1|O2]: p=2.765e-07
a.statistic.default(sample1 = f$ATP.Mut.Score.O1, sample2 =
f$ATP.Mut.Score.O2)
A statistic = 1
95% confidence interval for A = [NA, NA]
99% confidence interval for A = [NA, NA]
```

M. Statistical Tests for Case Study 3

The non-paired Wilcoxon signed-rank test was used for comparing the abstract and complete test models. The following abbreviates are used in the tests:

- AT: the all transitions coverage criterion
- RTP: the all round-trip paths coverage criterion
- ATP: the all transition pairs coverage criterion
- Complete: the detailed test model
- Abstract: the test model where contents in super states were removed
- O1: the state-invariant oracle
- O2: the state-pointer oracle
- Prep: the time spent on preparing the test suite
- Exec: the time spent on executing the test suite
- Mut. score: the mutation score; that is, the number of non-equivalent mutants killed divided by the total number of non-equivalent mutants

The following non-paired Wilcoxon signed-rank test was run:

```
wilcox.test(x, y, paired=FALSE, conf.level=0.95)
a.statistic.default(x$strategy.measure.test model.oracle,
x$strategy.measure.test model.oracle)
```

Preparation time

The abstract test model versus the complete test model using the state-invariant oracle

```
H_0: ABSTRACT_O1_PREP.TIME = COMPLETE_O1_PREP.TIME (non-paired)
data:  x$ABS.PREP.TIME.O1 and x$COMP.PREP.TIME.O1
W = 930, p-value < 2.2e-16
```

The abstract test model versus the complete test model using the state-pointer oracle

```
H_0: ABSTRACT_O2_PREP.TIME = COMPLETE_O2_PREP.TIME (non-paired)
data:  x$ABS.PREP.TIME.O2 and x$COMP.PREP.TIME.O2
W = 930, p-value < 2.2e-16
```

The abstract test model versus the complete test model using the state-invariant oracle and the state-pointer oracle

```
H_0: ALL_ABSTRACT_PREP.TIME = ALL_COMPLETE_PREP.TIME (non-paired)
data:  x$ABS.PREP.TIME and x$COMP.PREP.TIME
W = 3720, p-value < 2.2e-16
A statistics:
a.statistic.default(sample1 = x$ABS.PREP.TIME, sample2 =
x$COMP.PREP.TIME)
A statistic = 0.1148148
   95% confidence interval for A = [0.086, 0.152]
   99% confidence interval for A = [0.078, 0.166]
```

Execution time

The abstract test model versus the complete test model using the state-invariant oracle

```
H_0: ABSTRACT_O1_EXEC.TIME = COMPLETE_O1_EXEC.TIME (non-paired)
data:  x$ABS.EXEC.TIME.O1 and x$COMP.EXEC.TIME.O1
W = 141, p-value < 2.2e-16
```

The abstract test model versus the complete test model using the state-pointer oracle

```
H_0: ABSTRACT_O2_EXEC.TIME = COMPLETE_O2_EXEC.TIME (non-paired)
data:  x$ABS.EXEC.TIME.O2 and x$COMP.EXEC.TIME.O2
W = 736.5, p-value < 2.2e-16
```

The abstract test model versus the complete test model using the state-invariant oracle and the state-pointer oracle

```
H_0: ALL_ABSTRACT_EXEC.TIME = ALL_COMPLETE_EXEC.TIME (non-paired)
data:  x$ABS_EXEC and x$COMP_EXEC
W = 2426.5, p-value < 2.2e-16

A statistics:
a.statistic.default(sample1 = x$ABS_EXEC, sample2 = x$COMP_EXEC)
A statistic = 0.07489198
95% confidence interval for A = [0.054, 0.104]
```

```
99% confidence interval for A = [0.048, 0.114]
```

Mutation score

The abstract test model versus the complete test model using the state-invariant oracle

```
H_0: ABSTRACT_O1_MUT.SCORE = COMPLETE_O1_MUT.SCORE (non-paired)
data:  x$ABS.MUT.SCORE.O1 and x$COMP.MUT.SCORE.O1
W = 0, p-value < 2.2e-16
```

The abstract test model versus the complete test model using the state-pointer oracle

```
H_0: ABSTRACT_O2_MUT.SCORE = COMPLETE_O2_MUT.SCORE (non-paired)
data:  x$ABS.MUT.SCORE.O2 and x$COMP.MUT.SCORE.O2
W = 44, p-value < 2.2e-16
```

The abstract test model versus the complete test model using the state-invariant oracle and the state-pointer oracle

```
H_0: ALL_ABSTRACT_MUT = ALL_COMPLETE_MUT (non-paired)
data:  x$All.Abstract and x$All.Complete
W = 5505.5, p-value < 2.2e-16

A statistics:
a.statistic.default(sample1 = x$All.Abstract, sample2 =
x$All.Complete)

A statistic = 0.1699228
95% confidence interval for A = [0.132, 0.217]
99% confidence interval for A = [0.121, 0.233]
```