Information and Software Technology 56 (2014) 890–910

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Empirical evaluations on the cost-effectiveness of state-based testing: An industrial case study

Nina Elisabeth Holt [a,*], Lionel C. Briand [b], Richard Torkar [c,d]

[a] Simula Research Laboratory, Martin Linges vei 17, 1325 Lysaker, Norway
[b] SnT Centre, University of Luxembourg, Luxembourg
[c] Chalmers and the University of Gothenburg, Gothenburg, Sweden
[d] Blekinge Institute of Technology, Karlskrona, Sweden

ARTICLE INFO

Article history:
Received 28 February 2013
Received in revised form 16 February 2014
Accepted 25 February 2014
Available online 5 March 2014

Keywords:
State-based testing
UML
Cost effectiveness
Automated testing
Empirical evaluation
Industrial case study

ABSTRACT

Context: Test models describe the expected behavior of the software under test and provide the basis for test case and oracle generation. When test models are expressed as UML state machines, this is typically referred to as state-based testing (SBT). Despite the importance of being systematic while testing, all testing activities are limited by resource constraints. Thus, reducing the cost of testing while ensuring sufficient fault detection is a common goal in software development. No rigorous industrial case studies of SBT have yet been published.
Objective: In this paper, we evaluate the cost-effectiveness of SBT on actual control software by studying the combined influence of four testing aspects: coverage criterion, test oracle, test model and unspecified behavior (sneak paths).
Method: An industrial case study was used to investigate the cost-effectiveness of SBT. To enable the evaluation of SBT techniques, a model-based testing tool was configured and used to automatically generate test suites. The test suites were evaluated using 26 real faults collected in a field study.
Results: Results show that the more detailed and rigorous the test model and oracle, the higher the fault-detection ability of SBT. A less precise oracle achieved 67% fault detection, but the overall cost reduction of 13% was not enough to make the loss an acceptable trade-off. Removing details from the test model significantly reduced the cost by 85%. Interestingly, only a 24–37% reduction in fault detection was observed. Testing for sneak paths killed the remaining eleven mutants that could not be killed by the conformance test strategies.
Conclusions: Each of the studied testing aspects influences cost-effectiveness and must be carefully considered in context when selecting strategies. Regardless of these choices, sneak-path testing is a necessary step in SBT since sneak paths are common while also undetectable by conformance testing.

1. Introduction

In practice, software testing is often conducted as a manual, *ad hoc* task, instead of as an automated and systematic procedure. As a result, testing is likely to be incomplete and costly when trying to achieve an adequate level of dependability. Since thorough software testing is an expensive task, reducing the cost of testing while ensuring sufficient fault-detection effectiveness, is a common goal in the industry. In order for companies to make the right trade-offs when deciding how to test their software, it is essential that they understand how various test strategies compare in terms of cost effectiveness.

One approach to software testing derives test cases from a behavior model of the system under test (SUT) and is referred to as model-based testing (MBT) [1]. MBT is not a new research area in software engineering [2] but empirical evidence about its cost-effectiveness, especially in industrial contexts, is very limited. However, in recent years, the level of interest regarding MBT in both industry and academia has been increasing. This is visible from the many reported academic studies [1,3–7] and industrial projects [8–11] on model-based testing. This suggests that there might be an increasing awareness of the benefits provided by MBT [1].

As one of several possible input models for MBT, state machines are widely used to specify the behavior of the most critical and complex components that exhibit state-driven behavior [12]. Many object-oriented methodologies recommend modeling such

* Corresponding author. Tel.: +47 95087972.
E-mail addresses: neh@accedo.no, ninaeho@simula.no (N.E. Holt).

http://dx.doi.org/10.1016/j.infsof.2014.02.011
0950-5849/© 2014 Elsevier B.V. All rights reserved.

components with state models for the purpose of test automation [13], which is referred to as state-based testing (SBT). This is in part due to the fact that the specification of a software product can be used as a guide for designing functional tests [14]. As stated by Offutt and Abdurazik [14], formal specifications represent a significant benefit for testing because they precisely describe what functions the software is supposed to provide.

In particular, such specifications enable the automatic generation of test cases and oracles using MBT tools. Tool support for MBT has improved in recent years, but most of the tools specifically target an application context and cannot be easily adapted to others. A number of tools have been developed to support MBT, e.g., [8,11,15–19]. However, all of them have at least one of the following drawbacks. Well-established standards for modeling the SUT are not supported, thus making it difficult to integrate MBT with the rest of the development process, which in turn makes the adaptation and use of MBT more costly. Often, tools cannot be easily customized to different needs and contexts. For instance, a tester may want to experiment with different test strategies to help target specific kinds of faults. Moreover, practical constraints can evolve, such as the test-script language a company works with, and such changes are often not easy to accommodate.

However, regardless of the applied tool, there are several challenges related to investigating fault-detection effectiveness, for example the possibly limited number of faults that are present in the SUT [12]. Previous work has addressed this problem by seeding artificial faults into correct versions of the SUT using mutation operators [20,21]. Although the results from various studies [22–24] have suggested that faults seeded using mutation operators may, under certain conditions, be representative of realistic faults, it is still necessary to study fault-detection effectiveness in more realistic settings, using real faults, if we wish to increase the external validity of the results. A few studies have partially used naturally occurring faults [25]. However, these constituted only a minor percentage of the total number of faults. The research presented in this paper was motivated by a lack of empirical evidence regarding the state-based testing of industrial software with realistic faults and, thus, complements studies conducted in artificial settings.

Another key challenge in software testing is how to define and automate the test oracle, to determine whether or not the system is behaving as intended during a test execution. Very few studies empirically evaluate test oracles in state-based contexts. This area deserves more research, since the cost and fault-detection ability of different oracles may vary substantially [13].

Yet another interesting area of SBT, which has hitherto been given little attention, is the possible benefit of increasing cost effectiveness by removing details from the test model. Several studies, e.g. [26,27], focus on lowering the cost of testing by reducing the size of the test suites, while preserving their original coverage. There are conflicting results on how this reduction influences the fault-detection ability of the test suites, in particular with respect to how rigorous the test criteria are. Heimdahl and George [26] expressed concern about using this technique on structural coverage, due to the possible loss in fault-detection. Whereas such test-reduction techniques are based on removing tests in a test suite that do not affect the achieved level of coverage, this paper instead 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 need to be generated and maintained, but savings would also result from a less detailed test model that requires less modeling and maintenance effort.

The need for credible empirical results regarding the cost effectiveness of SBT leads to the central purpose of this paper: To empirically evaluate the cost effectiveness of SBT within the context of a safety–critical system by configuring and applying an extensible tool for automating the test procedure according to various strategies and oracles. More specifically, we will investigate the effect of coverage criteria, test oracle, test model, and sneak-path testing on cost effectiveness. Details regarding the requirements, design and development of the tool can be found in [28].

We used the embedded case study method [29] to evaluate these four aspects of SBT and their influence on cost effectiveness. The case study was conducted within the context of a research project at ABB, where a safety-monitoring component in a control system was developed using UML state machines [30] and implemented according to the extended state-design pattern [31].

In order to evaluate SBT techniques, an MBT tool, the TRansformation-based tool for Uml-baSed Testing (TRUST), was configured and used to automatically generate the studied test suites. The generated test suites were evaluated using 26 real faults collected in a global field study. The field study included 11 developers from three ABB departments who solved a maintenance task, split in six sub tasks, for the safety-monitoring component under study. Manual code inspections were used to collect actual faults.

Results from the case study indicate that the evaluation of coverage criteria regarding fault-detection are consistent with the results obtained using artificial faults in existing research, thus increasing the external validity of those results. We used two oracles of different precision levels: (1) the state-invariant oracle checks the state invariant of the resulting state; (2) the state-pointer oracle only checks that the array holding a pointer to the current state is as expected. Applying the most rigorous oracle, i.e., the combination of the state invariant and state pointer oracle, appears to be worthwhile given the significant increase in fault-detection effectiveness as compared to applying the state-pointer oracle alone. Interestingly, removing a rather substantial part of state-machine details resulted in a significant cost reduction (85% across all six criteria for both oracles), at the expense of an average reduction in fault-detection of 24–37% (depending on the applied oracle). Such results warrant further research into this type of test-suite reduction strategy. The application of sneak-path testing, i.e., testing triggers that should not result in any transition, also highlights the importance of this type of testing: All the mutants that remained undetected after applying conformance testing (i.e., checking that the system under test reacts according to the specified behavior) were killed by the sneak-path tests.

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

The findings from this case study are relevant for both industry and academia. Research on software testing that is to be adopted in industrial settings must provide evidence that it is cost-effective for the industry. For this, case studies are essential, in that they give the opportunity to test concepts in a real environment. In addition, the effort required by keeping state machine models updated may be easier for practitioners to accept when there is evidence for the cost effectiveness of SBT. Finally, the directions for future work should be useful as guidance for further research on SBT.

The remainder of this paper is structured as follows. Section 2 provides background information about SBT, test oracles, test models, and MBT tools. The research methodology is described in Section 3, whereas the execution of the study is described in Section 4. Section 5 reports on the results which are analyzed in Section 6, and further discussed in Section 7. Threats to validity are addressed in Section 8. Finally, Section 9 concludes the paper and suggests future work.

2. Background

2.1. Concepts

State-based testing (SBT) derives test cases from state machines that model the expected system behavior. The system under test (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 when the SUT is in that state.

Previous work on SBT has used coverage criteria that were defined to cover finite state machines. However, as an extension to a finite state machine, those criteria also apply to UML state machines if concurrency and hierarchy are removed [12]. The descriptions of the coverage criteria used in the following paragraph are based on definitions given by Binder [32], pp. 259–266 and Offutt et al. [33].

All transition coverage (AT) is obtained if every specified transition in a state machine is exercised at least once [32]. The order of the exercised transitions is not important. Applying this criterion ensures that all states, events and actions are exercised, and is considered as being the minimum amount of coverage that one should achieve when testing software [32]. *All round-trip* coverage (RTP) demands that all paths in a state machine that begin and end with the same state 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 a breadth- or depth-first traversal of the state machine. The test tree that corresponds to the RTP 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 begin with the initial state and end with the final state). According to Binder [32], 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. A weaker form of the RTP exercises only one of the disjuncts in guard conditions. *All transition-pairs* coverage (ATP) 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. A test suite that achieves *full predicate* coverage (FP) ensures that each clause in each predicate on guarded transitions is tested independently [14]. The *Paths of length n* are all possible sequences of transitions of length n from the initial state.

The above-mentioned test strategies are known as *conformance testing*, which seek to compare explicitly modeled behavior to actual software execution. However, it is also important to test whether or not the software correctly handles unspecified behavior, that Binder refers to as *sneak paths* [32]. State machines are usually incompletely specified and this is normally interpreted as events for which the system should not react, i.e., changing states or performing actions. Sneak-path testing sends every unspecified event in all states. In other words, its aim is to verify the absence of unintentional sneak paths in the software being tested as they may have catastrophic consequences in safety–critical systems.

One approach to evaluate the cost effectiveness of SBT strategies is *mutation analysis*. It is carried out by seeding automatically generated faults into "correct" versions of the SUT; one fault is seeded in each mutant version to avoid interaction effects between faults [12]. Mutants are identified via static analysis of the source code by a mutation system, such as in [34]. 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 to assess the test suite's fault-detection ability. 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.

2.2. Related work

Some of the existing MBT research has evaluated state-based coverage criteria within the context of UML state machines. The most studied state-based coverage criteria are FP, ATP, RTP, and AT. The FP criterion tends to obtain higher or similar mutation score as ATP, although at a higher cost. With this in mind, the AT, RTP, and ATP coverage criteria were selected for use in this paper.

AT has been found *not* to provide an adequate level of fault detection [12]. With the exception of results reported in [35], where only 54% of mutants were killed, ATP has shown to be a rather strong coverage criterion as compared to AT and RTP, although at a higher cost. RTP was shown to be more cost-effective than AT and ATP [12]. Another study [13] found that RTP testing is not likely to be sufficient in most situations as significant numbers of faults remained undetected (from 10% to 34% on average) across subject classes. This is especially true when using the weaker form of RTP. In [36], RTP was compared to random testing. The study concluded that the RTP strategy is reasonably effective at detecting faults; 88% of the faults were detected, as compared to 69% for random testing. Moreover, their results showed that RTP left certain types of faults undetected, and like in [13], it was 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. Several other studies also focus on combining test strategies; in [37,38], RTP was combined with white-box testing, resulting in significantly better fault-detection.

A study by Briand et al. [39] was conducted that aimed at investigating how data-flow information could be used to improve the cost effectiveness of RTP when more than one transition tree could be generated. The results showed that data flow information was useful for the selection of the most cost effective transition tree. The RTP criterion was further studied by Briand et al. [40] with a 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" [40]. The results suggested that data flow information in a transition tree could be used to select the tree with the highest fault-detection ability.

The majority of the results found in existing SBT research are based on studies where mutation operators have been applied to small, non-industrial programs, like the well-known Cruise Control system example [41], p. 595. With the exception of two studies where only a small percentage of the seeded faults were real [14,25], artificial faults were overwhelmingly used in the reported testing strategy evaluations [12,13,36,42,43]. The few studies that partially use realistic faults, e.g., [14,25], often fail to mention how those faults were collected. As stated by Andrews et al. [24], a problem when evaluating testing strategies is that real programs with realistic faults are rarely available. As a consequence, little is known about how such structured test approaches compare when detecting realistic faults. When also considering the setting of the experiment, only two studies [43,44] were executed in industrial settings. Of these two studies, the first study [43] did not report the nature of the seeded faults, whereas the second study [44] reported the use of mutation operators. The two studies [14,25] that actually did report the use of realistic faults were conducted in laboratory settings, and in those cases only 4/20 and 4/21 of the faults were real.

Very few studies have compared oracles within the context of SBT; in fact, the study of Briand et al. [13] appears to be unique in its empirical comparison of oracles within this particular context. Although not being the sole focus of their study, the results revealed statistically significant differences between the two oracle strategies. They found that the precise oracle had significantly stronger fault-detection abilities than the state-invariant oracle but led to increased cost: Since less attributes are checked, the state-invariant oracle required less resources during both development and execution. Fault-detection went from 50% when only checking the state invariant to 72% when also checking contract assertions.

Reducing the test-suite size by abstracting the test model is yet another area of related work where there is a lack of research within the context of SBT. Several studies, in other contexts, focus on lowering the cost of testing by reducing the test suites while preserving the original coverage, but at the expense of fault-detection ability. Heimdahl and George [26] found that the size of the specification-based test suites can be dramatically reduced but that, as a result, the fault detection of the reduced test suites is adversely affected. Wong et al. [27] investigated the effect on fault-detection of reducing the size of a test suite while keeping block and all-uses coverage constant. They found that the reduction in effectiveness was not significant, even for the most difficult faults, which suggests that the minimization of test suites can reduce the cost of testing at a slightly reduced level of fault-detection effectiveness. Similar to the work in [26], we investigated the fault-detection effectiveness of reduced test suites, but based on a different idea. Whereas test-reduction techniques are based on removing tests that do not contribute to an increase in coverage for the test suite, this paper focuses instead on abstracting the test model itself. This means that not only are the number of test cases in the test suites reduced, but the detail level in the test model is reduced as well, thus reducing its construction and maintenance cost.

The final aspect of SBT is sneak-path testing. Mouchawrab et al. [38] conducted a series of controlled experiments for the purpose of investigating the impact of RTP on cost and fault detection when compared to structural testing. The study was a replication of the experiment in [13] where one of the findings was that not testing sneak-path transitions 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 [32]. The 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 such studies in realistic industrial contexts. Part of the study reported here, regarding sneak-path testing, has been previously published [45].

To conclude, though some reported research has evaluated state-based coverage criteria, the focus has mostly been directed towards fault-detection effectiveness based on artificial seeded faults (e.g., with mutation operators) and was not concerned with the cost of such testing. In addition, the majority of the results are based on studies involving small academic programs. The few studies that partially use realistic faults [14,25] tend not to describe how those faults were collected. Our study complements and extends existing research on the cost effectiveness of SBT by:

- using an industrial safety–critical control system as subject,
- using real faults (collected from an industrial field study),
- comparing six state-based coverage criteria,
- comparing two test oracles at different levels of precision,
- studying the impact of removing details from the test model, and
- applying sneak-path testing.

3. Case study design

This section describes how our empirical evaluations on the cost effectiveness of state-based testing (SBT) were carried out. Studying multiple aspects within the same context makes this a single embedded case study [29].

3.1. Research objectives

The main purpose of this study was to investigate the influence of four aspects of cost effectiveness in SBT: six state-based coverage criteria, two different oracles, two test models at different levels of detail, and sneak path-testing.

The following research questions were addressed:

- *RQ1:* What is the cost and effectiveness of the following 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 cost and effectiveness?
- *RQ3:* What is the influence of the test model abstraction level on cost and effectiveness?
- *RQ4:* What is the impact of sneak-path testing on cost and effectiveness?

3.2. Case and subject selection

The case study was conducted within the context of a software process improvement project at ABB. ABB is a global company that operates in approximately 100 countries with 135,000 employees. Its primary business focus is automation technologies, for which the company develops software and hardware.

The current project was initiated to investigate which UML diagrams may be beneficial to ABB, especially during 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, state machine-driven UML modeling and to evaluate state-based testing in a realistic safety–critical development environment.

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 for supervising industrial machines: the safety board. 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 part of this system, called the Application Logic Controller (ALC). The main function of this 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 should also interface with an optional safety bus to enable a remote stop and a reliable exchange of system status information. The system is configurable with respect to which safety buses the system can interact with. This `ALC` subsystem was chosen for the 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 is the main source in the process of deriving automated test oracles.

3.3. Data collection procedures

To measure the cost and effectiveness of SBT, four surrogate measures were used, inspired by the study of Briand et al. [13].

Cost is measured in terms of:

- Test-suite preparation time. The time spent on generating the test tree, generating the test suite, and building the test suite.
- Test-suite execution time. The time spent on executing the test suite, as measured by completion time minus the time where inputs from external devices are simulated.
- Test-suite size. The number of test cases in a test suite.

Effectiveness is measured by:

- Mutation score. The number of non-equivalent mutants killed divided by the total number of non-equivalent mutants.
- 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.

A Technical Requirements Specification, developed by the business unit in cooperation with ABB scientists, was the starting point for developing a common understanding of the system. The modeling was a cooperative activity between the researchers and ABB. Each modeling activity was closely monitored. The control system was implemented as a joint effort between ABB and Simula researchers. The testing, however, was exclusively performed by the first author of this paper.

4. Case study execution

Motivated by the lack of extensible and configurable model-based testing (MBT) tools, Ali et al. [28] proposed an MBT tool, the TRansformation-based tool for UML-baSed Testing (TRUST), to be used in conjunction with other tools. The software architecture and implementation strategy of TRUST facilitated its customization for different contexts by supporting configurable and extensible features such as input models, test models, coverage criteria, test data generation strategies, and test script languages.

The remainder of this section describes how data collection was conducted. The main activities include the preparation of test models (Section 4.1), collecting fault data for creating mutants (Section 4.2), extending and configuring the testing tool TRUST (Section 4.3), and the generation and execution of test suites (Section 4.4).

4.1. The preparation of test models

The original model was tested using TRUST and the round-trip path (RTP) criterion. However, due to limitations in TRUST, some adjustments had to be made to the original model before it could be used as an input for TRUST. Since the flattening component does not support transitions that cross state borders, these transitions had to be re-modeled to and from the super-state edges. This also required adding initial states, entry points, and exit points in several of the super states. The state-machine flattener does not support multiple events on a single transition. Such transitions were thus transformed to one transition per event. Note that the applied changes did not affect the functionality as originally modeled.

The implemented flattening 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 [46]. The key aspects in these rules address (1) how to combine concurrent states, and (2) how to redirect transitions. Interested readers may consult [47] for more detailed information about the flattening algorithm and its corresponding transformations and implementation.

After executing 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. TRUST was executed with the configuration values presented in Table 1 and the flattened state machine described in Table 2 as an input. In this case, TRUST was configured for the RTP criterion, applied on a test tree, which conforms to the test tree metamodel presented in [28].

The test suite was then executed on what is considered to be a “correct” version of the code, i.e., one that does not cause the test suites to detect failures. Results were 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, and then re-generating the concrete test cases. An example of such an externally controlled variable is the *enable device*, which is managed by the operator and used for starting and stopping the movement of industrial machines.

Furthermore, even when a system is carefully specified and implemented, there is always the risk of introducing discrepancies between the specification and the implementation. Minor inconsistencies between the specification and the original version of the SUT were found during RTP testing – these inconsistencies, however, had to be resolved in order to run the tests without errors before moving on with the study.

To enable the evaluation of SBT strategies, a field study was planned for the purpose of collecting real fault data to be used in the generation of realistic mutants. For this reason, the original version of the system was modified to include a fourth operation mode: the *ExtraSlow* mode. Figs. 1 and 2 illustrate the change at a high level.

The UML state machine consisted 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, 6 simple-composite states, and a maximum hierarchy level of 2. The unflattened state machine contained 61 transitions. Having both concurrent

Table 1
Configuration parameters of TRUST.

Parameter	Value
Input model	UML2.0 state machine
Test model	Test tree for round trip paths
Coverage criterion	All round-trip paths
Oracle	State invariant + state pointer
Test scripting language	C++
Test data generation technique	Random data generation
OCL evaluator	EOS

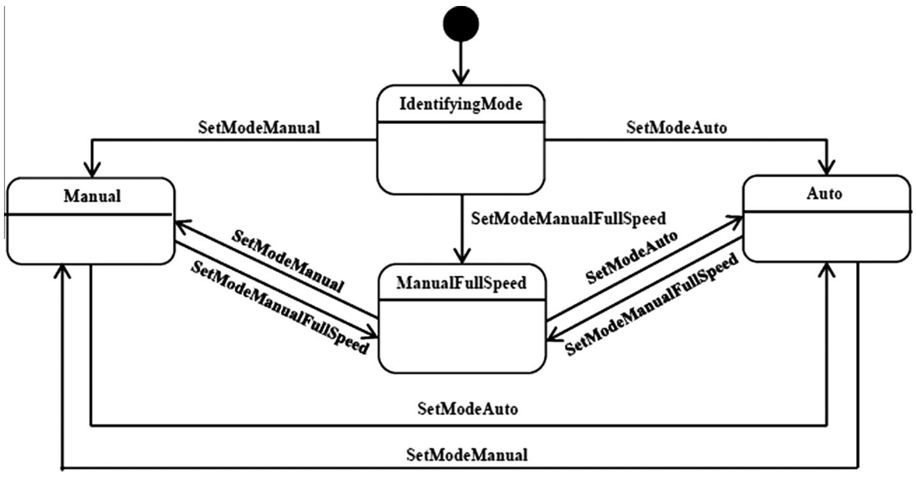

Fig. 1. The `Mode` region in the SUT prior to change task.

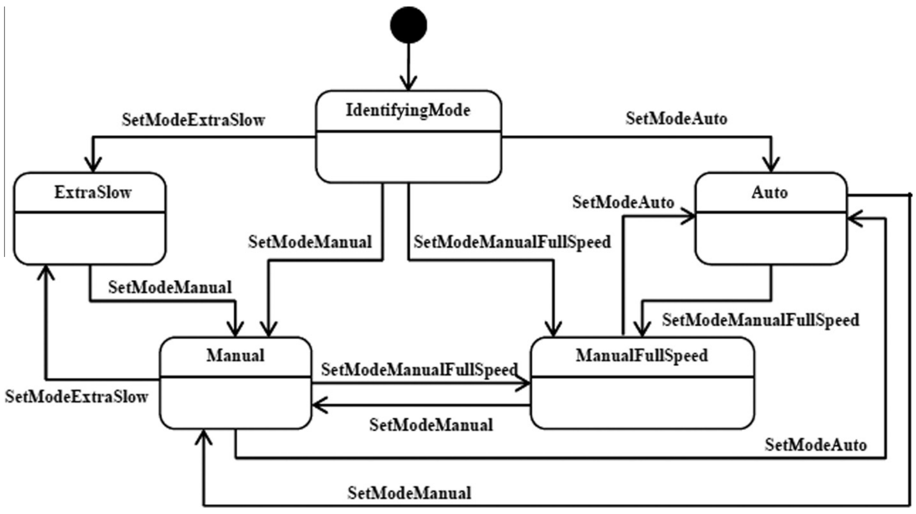

Fig. 2. The `Mode` region in the SUT after implementation of the change task.

Table 2
A features summary of the hierarchical scale of state machines – the original version of the SUT.

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

Table 3
A features summary of the hierarchical scale of state machines – the modified version of the SUT.

State-machine feature	Unflattened	Flattened
Maximum level of hierarchy	2	–
Number of submachines	0	–
Number of simple-composite states	6	–
Number of simple states	17	68
Number of orthogonal states	1	–
Number of transitions (guarded)	61 (17)	349 (107)

and hierarchical states, the state machine had to be flattened before being used as an input for test case generation. For this purpose, the state-machine flattening component of TRUST was used. On 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 state-machine flattener does not automatically remove these. The user can specify preferences in the provided Kermeta [46] transformation. The outcome of the transformation is a model where these combinations are excluded. After removing infeasible transitions, due to both incompatible state invariants (12 state combinations, 145 transitions) and guards that cannot evaluate to true (14 transitions), the state machine included 68 simple states (excluding initial and final state) and 349 transitions, of which 107 were guarded. The characteristics of the unflattened and flattened UML state machines are summarized in Table 3.

To observe 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 level in the state hierarchy, the model was abstracted to generate a less precise but correct test model. The contents of every simple composite state were removed. Note that, from now on, this model is referred to as the *abstract model*.

Raising the abstraction level prompted 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 (see Fig. 3). This means that the transitions that were targeted in entry points or sourced in exit points contained in the super-states were deleted. This is due to the fact that those transitions do not capture a common behavior to all sub-states of that super-state; such behavior is only common to those sub-states that have incoming transitions to a particular entry point. The same regards outgoing transitions from exit points. Consequently, parts of the super-state behavior are overlooked. Transitions sourced in the edge of a super state, on the other hand, concern all sub states.

In order to potentially capture more of the SUT's behavior, the transitions sourced in exit points belonging to the super-state could also be kept. This means that more of the possible behavior is tested, although 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. Although not impossible, it is difficult to know upfront whether or not the guard can be satisfied. It would be applicable if an analysis of the required path was implemented in order to satisfy the guard.

The outcome of applying the previously described abstraction approach 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, this model also 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 state and 86 feasible transitions, of which 28 were guarded. The characteristics of the unflattened and flattened UML state machines are summarized in Table 4.

The SUT was implemented according to the extended state-design pattern [31]. The extended state-design pattern provides a template on how to implement a UML state machine in such a way as to improve traceability from the model to the implementation and facilitate change. It extends the original state pattern by specifying how to implement state and transition actions. The extended state-design pattern localizes state-specific behavior in an individual class for each state and, hence, puts all the behavior for that state into one class. The pattern allows a system to change its behavior when its internal state changes. This is accomplished by letting the system change the class representing the state of the object. Fig. 4 shows an abstracted version of the class diagram of the SUT. It consists of a context class `ApplicationLogicController` (`ALC`) that represents the system controlling the machine. The `ALC` is assigned all responsibilities for handling the state behavior. The `ALC` is always in two concurrent states. In the `Mode` region, the `ALC` can be in `IdentifyingMode`, `Auto`, `Manual`, or `ManualFullSpeed` (`ManualFS`); in the `DriveEnable` region, the `ALC` can be in `Init`, `Enabled`, `Disabled` or `Halt`. The context class represents the system and its interface with the external environment. The `ALC` will act differently depending on what state it is in. Two abstract classes, `Mode` and `DriveEnable`, represent the states of the `ALC`, and declares interfaces common to all classes that represent different operational states. The subclasses of the abstract classes implement state-specific behavior, which means that they provide operation implementations that are specific to every state (and possibly empty) of the `ALC`. Experiences and lessons learned from applying the extended state design pattern can be found in [48]. The resulting C++ code for the SUT consisted of 26 classes and 3372 LOC (1227 h, 2145 cpp) (without blank lines).

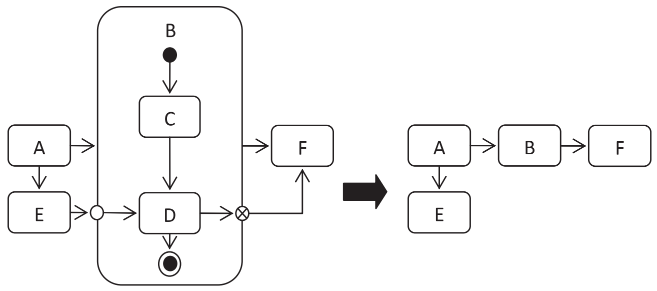

Fig. 3. Removing details from the test model.

Table 4
A features summary of the hierarchical scale of state machines – the abstract version of the SUT.

State-machine feature	Unflattened	Flattened
Maximum level of hierarchy	–	–
Number of submachines	–	–
Number of simple-composite states	–	–
Number of simple states	25	21
Number of orthogonal states	1	–
Number of transitions (guarded)	123 (35)	86 (28)

4.2. Collecting fault data for creating mutants

For the purpose of evaluating the cost effectiveness of SBT and to increase the external validity of the results, actual fault data was collected in a global field study to generate mutated versions of the SUT. The field study was conducted at ABB's offices in Västerås, Baden, and Shanghai. Having varied UML and domain knowledge, 11 ABB engineers were asked to implement a change task to the modified version of the SUT. The change task was suggested by ABB as a realistic modification. They were instructed to modify both the model and code. Participants were instructed to work strictly individually. One researcher supervised the sessions.

The maintenance task itself was initially suggested at a high level by ABB; however, it was defined and split into six sub tasks by the researchers, and finally approved by the company. The maintenance task consisted of adding an extra gear or mode, *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 textual and graphical documentation, in addition to a manual on how to apply the design pattern.

A version of the model and code, representing the SUT after the implementation of the maintenance task, was developed by one researcher and tested with each of the coverage criteria within the focus of this study in order to verify its correctness. The faults collected from the field study, described below, were inserted into this correctly modified version of the SUT to generate mutants.

Manual code inspections of the ten[1] solutions produced in the field study were used to collect actual faults. In total, 26 faults were detected during the code inspections. This is a low number of faults

[1] 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.

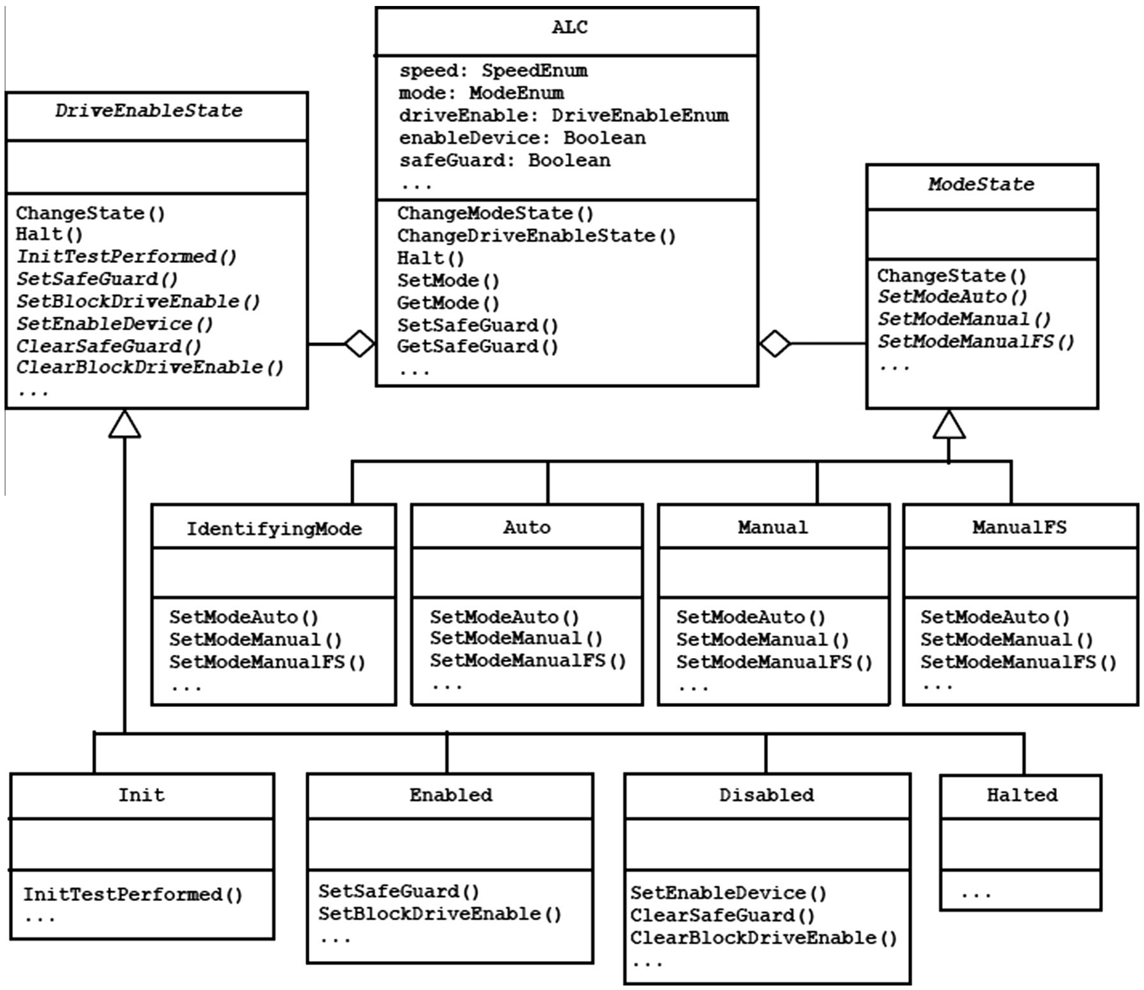

Fig. 4. Class diagram of the ALC.

compared to artificial mutation testing. However, real faults are rarely used in the reported testing strategy evaluations. As a consequence, little is known about how such structured test approaches compare in detecting real faults. In this paper, however, the comparison is exclusively based on real faults. These faults were introduced in both the model and code or only the latter. Note that because the objective was to compare the fault-detection abilities of the testing criteria, it was crucial that the seeded faults did not cause compilation errors. This means that only logical faults that could not be detected by the compiler could be selected. The extracted fault data were used to create 26 faulty versions (mutants) of the code by seeding one fault per program. The faults in the source code reflected the following modeling errors:

- *Missing transitions* (Fig. 5): An expected guarded transition triggered by a completion event from `State A` to `State B` was missing from the model. One of the participants only accounted for the transition that was explicitly triggered by the `e1()` event. The participant did not consider the transition that would fire if `g1` was already false.
- *Additional transitions* (Fig. 6): Another participant erroneously added transitions from `State C` not only, as specified, to `State A`, but also to `State B` and `State D`, and vice versa.
- *Guards that were not correctly updated* (Fig. 7): A participant added a clause in the guard on the transition from `State G` to `State H` so that a transition would be fired only if the mode is in `State C` or the speed is `extraSlow` in the concurrent region.
- *Guards that were modified which should not have been changed* (Fig. 8): In one of the erroneous versions, the event handling operation `e2()`'s guard was modified so that the state machine would transition from `State I` to `State J` only if the concurrent region was in `State C`.

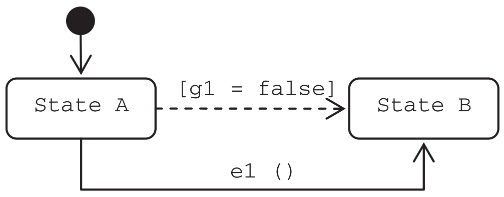

Fig. 5. Missing guarded transition.

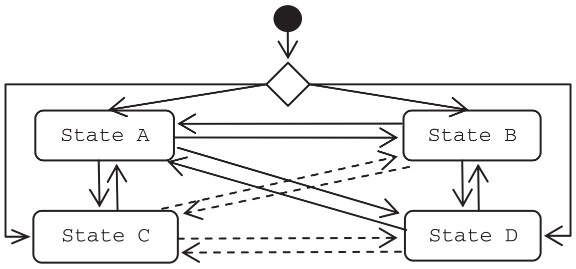

Fig. 6. Additional transitions to and from `State C`.

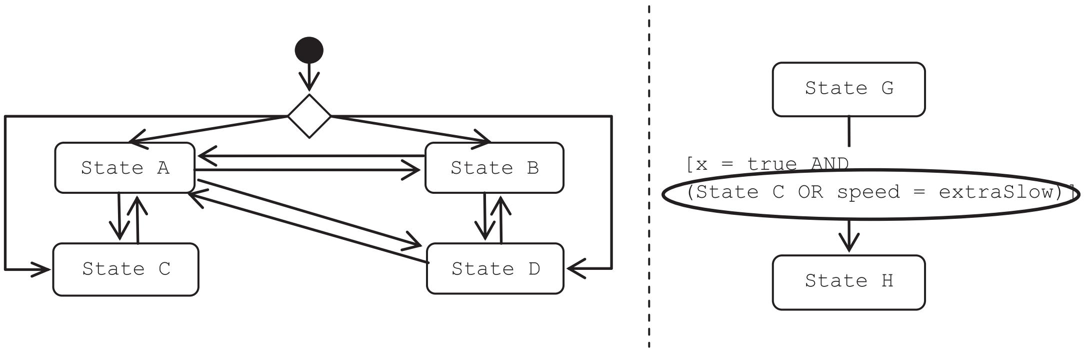

Fig. 7. Incorrect guard on transition from `State G` to `State H`.

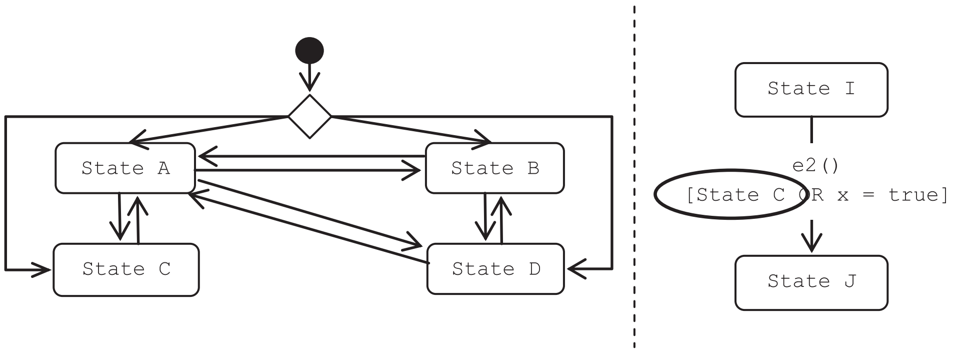

Fig. 8. Incorrect guard on transition from `State I` to `State J`.

- *Erroneous on-entry behavior* (Fig. 9): The on-entry behavior of `State K` was introduced with an error; the sub states, `State L` and `State M`, were updated with the same values for state variables. The only common value, however, should have been variable x. Variable y should have been given different values in the two sub states.
- *Incorrect state invariants:* State variable x was missing from `State C`'s state invariant.
- *Missing on-entry behavior:* `OnEntry()` was missing for `State C` (super-state).

Among these faults, eleven were sneak paths. To detect sneak paths, 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, only 15 out of 26 mutant programs were detectable by the conformance test suites (AT, RTP, ATP, LN2, LN3, and LN4).

4.3. Extending and configuring the testing tool TRUST

In order to use the model-based tool TRUST, it had to be extended and configured to meet the requirements and conditions of the investigation of this study. 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 the selection of values for externally controlled variables in guard conditions. Finally, TRUST was extended with support for sneak-path testing.

4.4. Generation and execution of test suites

TRUST was used to automatically generate executable test suites by model transformations. The prepared test model, previously introduced as the modified version of the SUT, was used as an input model for TRUST. As the state-based criteria are defined for finite state machines, a prerequisite for generating the test suites was to use an input state machine without concurrency and hierarchy. The first step in TRUST was thus to flatten [32] the test model, i.e., remove hierarchy and concurrency from the model, as previously described. The flattened state machine was then transformed into test trees by a set of ATL [49] transformation rules. 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 to the flattened state machine.

To illustrate the scope of the criteria, the following sequences of transitions are generated from the example model shown in Fig. 10:

- *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:* {(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)}

The generated test trees were input to the next transformation, which generated executable test cases. In this transformation,

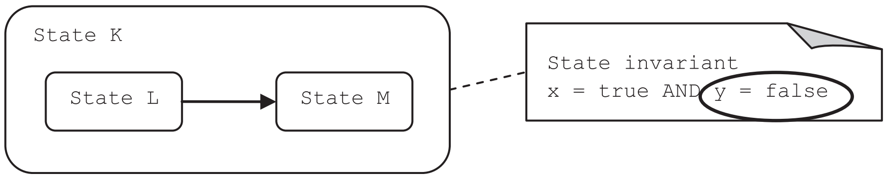

Fig. 9. Erroneous on-entry behavior that set state variables.

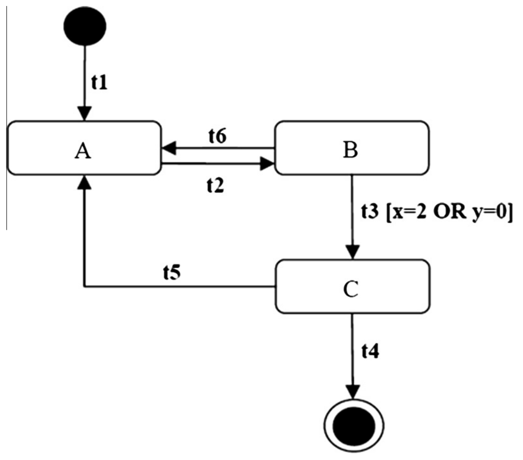

Fig. 10. Example state machine.

TRUST created concrete test cases using MOFScript[2] [50], which took the flattened state machine in addition to the generated test tree as an input. 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 (in our study, we used C++). Each path in the tree produces one test case. That is, 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. 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 on a new instance of the SUT.

The test suites were first executed on what was considered a "correct" version of the code; that is, one that does not cause the test suites to detect failures. The 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 the transitions to be fired, and then re-generating the concrete test cases. The environment is controlled by the test-data values. 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 [51,52], which are applicable in different situations. 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., [53]). In this study, however, the dynamic test-data generation was hard coded due to limited time resources. When the test suites executed successfully on the "correct" version, they were then run on the mutant versions of the SUT.

A batch file was created for each test strategy to automate the build, execution, and time data collection when executed on the correct and mutated programs. The version of the SUT to be executed was copied into a Visual Studio project folder. The project was then rebuilt and executed. One result text file was created for each test strategy. The result file contains the results of the correct version and the 26 mutants. Note that the conformance test suites were only run on the mutants that were not based on sneak paths. The rationale for this decision is simply that it is impossible to detect sneak paths with conformance test suites.

In summary, the following generation and execution steps were automated:

1. Flatten input state machine.
2. Select test adequacy criterion.
3. Construct abstract test cases in the form of a test tree. The algorithms used to traverse the state machine were described previously.
4. Select oracle.
5. Traverse the tree and select test data to generate concrete test cases. One test suite is generated per tree.
6. Build and execute the test suite on the correct version.
7. Ensure that the test suite is executed with no errors.
8. Then build again and execute the test suite on each of the mutant programs.
9. Analyze test results provided by the selected test oracle on all mutants.

For certain coverage criteria, like AT, RTP, and ATP, the generation of test trees from state machines is not deterministic, since several test trees could possibly satisfy the criterion. The structure of the tree depends on the sequence of the selected outgoing transitions when traversing the state machine as illustrated by Figs. 11 and 12. There are two types of dissimilarities: symmetric and semantic. Only the latter type has impact on the test results. The

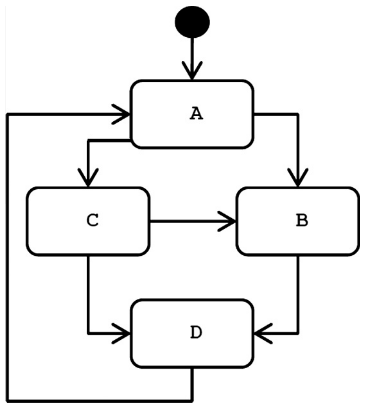

Fig. 11. Example state machine.

[2] A transformation language that generates text from models by following model-to-text transformation rules.

trees in Fig. 12 were generated by following the round trip path criterion using breadth first traversal. Even though the four trees differ symmetrically, Tree 1 is semantically equal to Tree 3 and Tree 2 is semantically equal to Tree 4. This implies that for example Tree 1 differs semantically from both Tree 2 and Tree 4. Thus, due to possible differences in fault-detection power, 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. Traditionally, in many fields of science, $n = 30$ has been a common rule of thumb to enable statistical testing [54]. We also complement our tests for significance with measures for effect size. Thus, 30 different trees were created using a random selection of outgoing transitions from states to generate distinct test suites. The test suites were obtained by traversing each of the 30 test trees and covering all paths. By selecting without a replacement from the population of all possible trees that achieves each of the criteria, only trees distinct from already selected trees were added to the selection.

5. Results

Tables 5–8 report the obtained results for each combination of coverage criteria, test oracle, and test model. The tables present descriptive statistics for each of the surrogate measures on cost and effectiveness: test-suite size (Table 5), test-suite preparation time (Table 6), test-suite execution time (Table 7), and mutation score (Table 8). Note that the state invariant oracle applied together with the state pointer oracle is referred to as oracle O1; the state pointer oracle applied in isolation is referred to as oracle O2.

The size of the sneak-path test suite is equal to the number of states in the SUT (68 simple states in the complete model and 21 simple states in the abstract model). Tables 9 and 10 show the time spent on preparing and executing a selection of test cases from the sneak-path test suite. We only measured preparation and execution time for the minimum number of sneak-path test cases that were required in order to kill all mutants. Table 11 shows realistic estimates for the entire sneak-path test suites. All eleven sneak path faults were detected by the sneak path test suite generated from the complete model. Due to an infeasible test case, the execution of the sneak-path test suite generated from the abstract model detected 10 out of 11 sneak paths.

6. Analysis

The analysis is divided into four parts: Sections 6.1–6.4 address each of the four research questions. For each research question, we present statistical tests on the data for AT, RTP, and ATP; and an analysis of cost and effectiveness. We use the $\widehat{A}_{12}$ statistic by Vargha–Delaney [55] to evaluate the effect size. Its range from 0 to 1 is divided in three categories: Small, medium, and large. Values of $\widehat{A}_{12}$ such that $0.36 < \widehat{A}_{12} \leqslant 0.44$ or $0.56 \leqslant \widehat{A}_{12} < 0.64$ indicate small effect size. Values of $\widehat{A}_{12}$ such that $0.29 < \widehat{A}_{12} \leqslant 0.36$ or $0.64 \leqslant \widehat{A}_{12} < 0.71$ indicate medium effect size. Values of $\widehat{A}_{12}$ such that $0 \leqslant \widehat{A}_{12} \leqslant 0.29$ or $0.71 \geqslant \widehat{A}_{12} \leqslant 1$ indicate large effect size. A value of 0.5 indicates no difference between the populations.

6.1. RQ 1: What is the cost and 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)?

In this section, we compare the cost effectiveness of six 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). The state-invariant oracle was applied together with the state-pointer oracle. Also note that test cases were generated from a complete test model.

6.1.1. Statistical tests

Tables 12–14 show the results from the paired Wilcoxon signed-rank tests that were performed to test for statistically significant differences in the 30 replications of AT, RTP, and ATP. The purpose of the tests was to reject the following null hypotheses:

$H_{0\text{-}cost}$: *There are no significant differences in cost when applying the testing strategies AT, RTP, and ATP on the complete model when combined with oracle O1.*

$H_{0\text{-}eff}$: *There are no significant differences in effectiveness when applying the testing strategies AT, RTP, and ATP on the complete model when combined with oracle O1.*

The statistical tests executed on preparation and execution time resulted in significantly different results – ATP spent significantly more time on both preparing and executing the test suites than AT and RTP. Effect sizes of the differences in preparation and execution time are also provided. Tables 12 and 13 present large effect sizes for all tests regarding costs; indicating practically significant differences in cost among test strategies. No significant differences were found in terms of mutation score though. Both RTP and ATP killed all mutants. AT killed all mutants in 29 out of 30 test suites – the final test suite killed 14 out of 15 mutants. The effect size of 0.5 shows a 50% probability of either strategy performing better.

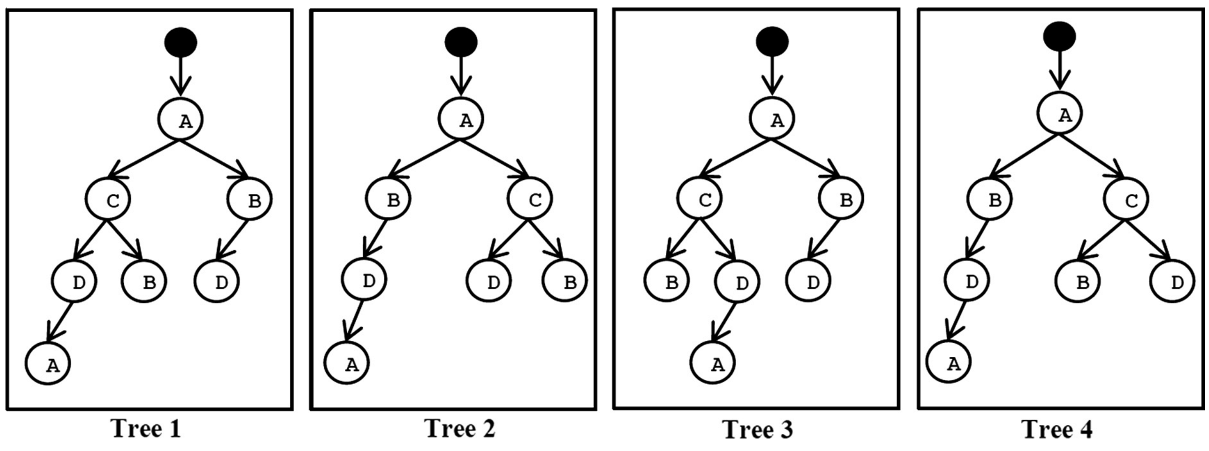

Fig. 12. Possible trees generated from Fig. 11.

Table 5
Test-suite sizes.

Coverage criterion	Test model	Oracle	Test-suite size total	Test-suite size feasible	Infeasible test cases (%)	Diff. between O1 and O2 (%)	Diff. between abstract and complete (%) (O1)	Diff. between abstract and complete (%) (O2)
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			
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			

Table 6
Descriptive statistics – preparation time.

Coverage criterion	Oracle	Model	Min (s)	Q1 (s)	Mean (s)	Median (s)	Q3 (s)	Max (s)	St. dev.	*N*	Diff. between abstract and complete (%)	Diff. between O2 and O1 (%) (abstract)	Diff. between O2 and O1 (%) (complete)
AT	O1	Abstract	173	180	222	192	210	972	143	30	−94.5	−3.8	−7.0
		Complete	2506	2763	3995	3013	3665	15,939	3264	30			
	O2	Abstract	168	175	213	185	202	965	143	30	−94.3		
		Complete	2249	2474	3715	2766	3394	15,663	3280	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	1089	1105	1115	1111	1123	1150	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	1088	1097	1141	1116	1161	1368	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	3943	3943	3943	3943	3943	3943	–	1	−25.5	1.3	3.8
		Complete	5295	5295	5295	5295	5295	5295	–	1			
	O2	Abstract	3993	3993	3993	3993	3993	3993	–	1	−27.3		
		Complete	5494	5494	5494	5494	5494	5494	–	1			

In short, ATP costs more than AT, which again costs more than RTP. Regardless of these differences, however, the three strategies yielded a similar mutation score. Thus, RTP can be considered to be the most cost-effective strategy when using the complete model and the combination of the state-invariant and state-pointer oracles.

Based on the obtained results, the null hypothesis was rejected for the surrogate measures for cost. However, no evidence was found to reject the null hypothesis with respect to effectiveness (mutation score).

6.1.2. Cost effectiveness analysis

This section focuses on the relationship between cost and effectiveness. Each of the surrogate measures of cost and effectiveness has been depicted in Fig. 13: Dark colors indicate high values; light colors represent low values. We observed that five of the criteria

Table 7
Descriptive statistics – execution time.

Coverage criterion	Oracle	Model	Min (s)	Q1 (s)	Mean (s)	Median (s)	Q3 (s)	Max (s)	St. dev.	N	Diff. between abstract and complete (%)	Diff. between O2 and O1 (%) (abstract)	Diff. between O2 and O1 (%) (complete)
AT	O1	Abstract	29	42	64	57	86	129	27	30	−97.4	−73.8	−83.1
		Complete	1765	2108	2455	2377	2785	3617	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	2607	2972	3341	3381	3669	3978	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			

Table 8
Descriptive statistics – mutation score.

Coverage criterion	Oracle	Model	Min	Q1	Mean	Median	Q3	Max	St. dev.	N	Diff. between abstract and complete (%)	Diff. between O2 and O1 (%) (abstract)	Diff. between O2 and O1 (%) (complete)
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			

provided a good mutation score – LN2 was the weakest. Looking at the cost for each strategy, ATP stands out with its high values for all cost measures. RTP and LN3 performed quite similarly. AT had the second highest cost, but a similar effectiveness to that of ATP, RTP, and LN4.

Despite killing all mutants, ATP is clearly the most expensive criteria to prepare. On the other extreme, LN2 has the lowest costs, but the fault-detection ability is correspondingly poor. There are significant differences regarding costs, e.g., preparation time versus execution time, between AT and RTP. Preparing the AT test suites

Table 9
Sneak-path test cases – preparation time and execution time – abstract test model.

Test case executed on mutant	Oracle	Time prepare test case (s)	Diff. O2 by O1 (%)	Time execute test case (s)	Diff. between O2 and O1 (%)
M18	O1	19	−10.5	4	−50.0
	O2	17		2	
M23	O1	19	−5.3	4	−50.0
	O2	18		2	
M24	O1	19	−10.5	4	−50.0
	O2	17		2	
M19	O1	21	−14.3	4	−50.0
	O2	18		2	
M25	O1	19	−10.5	3	−33.3
	O2	17		2	
M16	O1	19	−10.5	5	−60.0
	O2	17		2	
M17	O1	20	−10.0	4	−50.0
	O2	18		2	
M20	O1	20	−15.0	4	−50.0
	O2	17		2	
M21	O1	18	−5.6	4	−50.0
	O2	17		2	
M22	O1	19	−10.5	5	−60.0
	O2	17		2	
M26	O1	20	−10.0	5	−60.0
	O2	18		2	

Table 10
Sneak path test cases – preparation time and execution time – complete test model.

Executed on mutant	Oracle	Time prepare test case (s)	Diff. between O2 and O1 (%)	Time execute test case (s)	Diff. between O2 and O1 (%)
M23	O1	25	0.0	7	−71.4
	O2	25		2	
M18	O1	26	−11.5	4	−50.0
	O2	23		2	
M24	O1	26	0.0	4	−50.0
	O2	26		2	
M19	O1	25	−12.0	6	−66.7
	O2	22		2	
M25	O1	25	−12.0	4	−50.0
	O2	22		2	
M16	O1	28	−25.0	3	−33.3
	O2	21		2	
M17	O1	25	−16.0	4	−50.0
	O2	21		2	
M20	O1	23	−8.7	5	−60.0
	O2	21		2	
M21	O1	34	0.0	6	−66.7
	O2	34		2	
M22	O1	34	0.0	4	−50.0
	O2	34		2	
M26	O1	34	0.0	4	−50.0
	O2	34		2	

took slightly more time than RTP, but less time than LN4. Looking at the execution time, Fig. 13 indicates that AT uses more time compared to both RTP and LN4.

Finding 1: The results for cost and effectiveness indicate that LN2 may be too weak as a testing strategy. The remaining testing strategies killed all mutants (except from sneak paths), but with varying costs. ATP was the most expensive criterion and RTP the least expensive. In our case, RTP is therefore the most cost-effective strategy, closely followed by LN3.

Table 11
Estimated preparation and execution times for sneak-path test suites.

Test model	Oracle	Preparation time (estimate in seconds)	Execution time (estimate in seconds)
Complete test model	Oracle O1	1885	315
	Oracle O2	1750	136
Abstract test model	Oracle O1	407	88
	Oracle O2	365	42

6.2. RQ2: How does varying the oracle affect cost and effectiveness?

This section extends the comparison of state-based coverage criteria by investigating the influence of varying the test oracle on cost effectiveness. It is important to remember the difference between the two oracles O1 and O2: the former checks the state invariant in addition to the pointer to the current system state, whereas the latter only checks the pointer to the current system state. Test cases were generated from the complete test model. Section 6.2.1 reports on the results of applying the statistical tests. Section 6.2.2 presents an analysis of the cost and effectiveness.

6.2.1. Statistical tests

The paired Wilcoxon signed-rank test was executed on the collected data when applying two different oracles on the complete model. The purpose of the tests was to reject the following null hypotheses:

$H_{0\text{-}cost}$: *There are no significant differences in cost when applying two different oracles for each of the AT, RTP, and ATP criteria on the complete model.*

$H_{0\text{-}eff}$: *There are no significant differences in effectiveness when applying two different oracles for each of the AT, RTP, and ATP criteria on the complete model.*

Varying the oracle shows that there were significant differences in the preparation time for AT (medium effect size) and ATP (large effect size); O1 required more preparation time than O2. For RTP, on the other hand, no significant differences between the two oracles were found. The execution time was significantly higher when applying oracle O1. Moreover, the effect size was large. This pays off, however, in that O1 consistently achieves a significantly higher mutation score. In this case, the effect size is also large.

As we can see from the results of applying the statistical tests (Tables 15–17), the null hypotheses were rejected for all strategies regarding effectiveness, but not for cost. Note, however, that the only null hypothesis regarding cost that could not be rejected was related to RTP's preparation time. For AT and ATP there are significant differences in cost and effectiveness when applying two different oracles on the complete model.

6.2.2. Cost effectiveness analysis

This section investigates the relationship between cost and effectiveness for SBT when varying the oracle. Fig. 14 graphically illustrates the relationship between cost and effectiveness by displaying the surrogate measures for cost[3], preparation and execution time, and the surrogate measure for effectiveness, mutation score.

[3] Recall that test-suite sizes were not influenced by varying the oracle when generating tests from the complete test model; hence, this surrogate measure is not addressed in this section.

Table 12
Paired Wilcoxon signed-rank test – 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
AT = ATP	O1	Complete	Prep. time	1.86e–09	1	Large	AT < ATP	Yes
RTP = ATP	O1	Complete	Prep. time	1.86e–09	0	Large	RTP < ATP	Yes

Table 13
Paired Wilcoxon signed-rank test – 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
AT = ATP	O1	Complete	Exec. time	4.50e–06	0.086	Large	AT < ATP	Yes
RTP = ATP	O1	Complete	Exec. time	1.86e–09	0	Large	RTP < ATP	Yes

Table 14
Paired Wilcoxon signed-rank test – 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	0.5	No effect	AT = RTP	No
AT = ATP	O1	Complete	Mut. score	NA	0.5	No effect	AT = ATP	No
RTP = ATP	O1	Complete	Mut. score	NA	0.5	No effect	RTP = ATP	No

Obviously, an ideal situation would have been low values for cost, but high mutation scores. However, this is not the case for all combinations of coverage criteria and oracles. In particular, we can observe from Fig. 14 that ATP combined with both oracles O1 and O2 is distant from the desired area. The results suggest that the mutation score is negatively affected for all coverage criteria when using the 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 can see from Fig. 14, a strong oracle combined with weaker coverage criterion clearly improves fault-detection ability.

Even though the two oracles achieved rather similar levels of cost in terms of preparation time (Fig. 14), results show that using oracle O1 resulted in an overall higher cost when compared to oracle O2. The highest impact was seen for LN2, followed by LN3. Larger differences were seen for cost when focusing on execution time (Fig. 15); oracle O2 required less time than oracle O1.

Finding 2: The combination of coverage criterion and oracle significantly impacts the cost (except from preparation time for RTP) and effectiveness. 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 significant amount of cost savings when using O2, however, had an overall negative impact on effectiveness.

6.3. RQ 3: What is the influence of the test model abstraction level on cost and effectiveness?

In Sections 6.1 and 6.2, we noted how six coverage criteria compare when applied to a complete test model combined with two oracles of different strengths. This section deals with the test model itself, especially with respect to the level of detail, and seeks to provide answers as to how a less detailed test model in input to the test case generation would affect cost effectiveness. The results are collected by generating test suites and running tests with a model where the contents of every composite state, in addition to the belonging transitions, were removed (abstract model).

6.3.1. Statistical tests

Due to the independence between test paths for each of the 30 replications of AT, RTP, and ATP in the complete versus the abstract test model, these cannot be considered to be pairs in statistical tests. Hence, the independent samples Wilcoxon signed-rank test was used for the statistical testing of differences. The purpose of the tests was to reject the following null hypotheses:

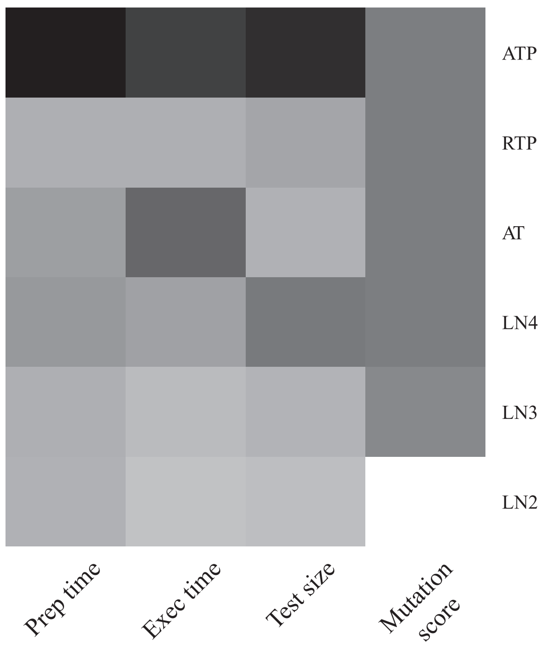

Fig. 13. Preparation and execution time, test-suite size and mutation score.

Table 15
Paired Wilcoxon signed-rank test comparing oracles O1 and O2 – preparation time.

H_0	Strategy	Measure	*p*-Value	$\hat{A}_{12}$	Effect size	95% CI for $\hat{A}_{12}$	Result	Sign. diff. (CI)
O1 = O2	AT	Prep. time	1.82e−06	0.67	Medium	[0.521,0.794]	O1 > O2	Yes
O1 = O2	RTP	Prep. time	1	–	–	[NA,NA]	O1 = O2	No
O1 = O2	ATP	Prep. time	0.033	0.72	Large	[0.573,0.835]	O1 > O2	Yes

Table 16
Paired Wilcoxon signed-rank test comparing oracles O1 and O2 – execution time.

H_0	Strategy	Measure	*p*-Value	$\hat{A}_{12}$	Effect size	Result	Sign. diff. (CI)
O1 = O2	AT	Exec. time	1.82e−06	1	Large	O1 > O2	Yes
O1 = O2	RTP	Exec. time	1.82e−06	1	Large	O1 > O2	Yes
O1 = O2	ATP	Exec. time	1.86e−09	1	Large	O1 > O2	Yes

Table 17
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
O1 = O2	RPT	Complete	Mut. score	4.61e−08	1	Large	O1 > O2	Yes
O1 = O2	ATP	Complete	Mut. score	2.76e−07	1	Large	O1 > O2	Yes

Table 18
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)
A = C	AT, RTP, ATP	O1	Prep. time	<2.2e−16	0.115	Large	A < C	Yes
A = C	AT, RTP, ATP	O2	Prep. time	<2.2e−16	0.115	Large	A < C	Yes

$H_{0\text{-}cost}$: There are no significant differences in cost for the strategies AT, RTP, and ATP when varying the level of details in the test models.

$H_{0\text{-}eff}$: There are no significant differences in effectiveness for the strategies AT, RTP, and ATP when varying the level of details in the test models.

Tables 18–21 provide results from the statistical tests. Overall, when considering the three strategies combined with each of O1 and O2, significant differences were found for all preparation times, execution times and mutation scores. The low *p*-values show that the null hypotheses can be rejected, and we can conclude that there are significant differences in both cost and effectiveness for the strategies AT, RTP, and ATP when varying the level of detail in the test model. Preparation times, execution times, and mutation scores are significantly higher when using the complete test model.

To summarize, 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. The results showed that ATP applied to a complete model is an expensive strategy. The high fault-detection effectiveness may come at too high a cost.

Combining ATP with the abstract model resulted in a significant cost reduction. The effectiveness was also reduced. On the other extreme, LN2 had the lowest cost but also the lowest effectiveness; 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 detail 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. RTP and LN4, when applied to the complete model and oracle O1, obtained as good mutation scores as ATP and AT. Of these, RTP had the lowest costs. Overall, the abstract model performs better with O1, the most precise oracle.

AT, RTP, ATP, and LN4 all provided the highest mutation scores when generated from a complete model used with oracle O1. Of these, RTP had the lowest costs. Using the state-pointer oracle, and 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.

6.3.2. Cost effectiveness analysis

This section discusses how the removal of details from the test model affects cost effectiveness. Fig. 16 displays the cost[4]-effectiveness of the 24 combinations of coverage criteria, test 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 represents oracle O2 and the complete test model; yellow represents oracle O1 and the abstract test model; and finally, red represents oracle O2 and the abstract test model.

It is worth examining the bottom of the figure on the left hand side. For LN2, we can, as predicted, observe that by reducing the level of details in the test model, both for oracles O1 and O2, the mutation score is lowered even further. Significant differences on both cost and effectiveness were observed for AT when increasing the abstraction level. None of the combinations of LN2 can be rec-

[4] Please note that time is shown as the sum total of the preparation time and execution time.

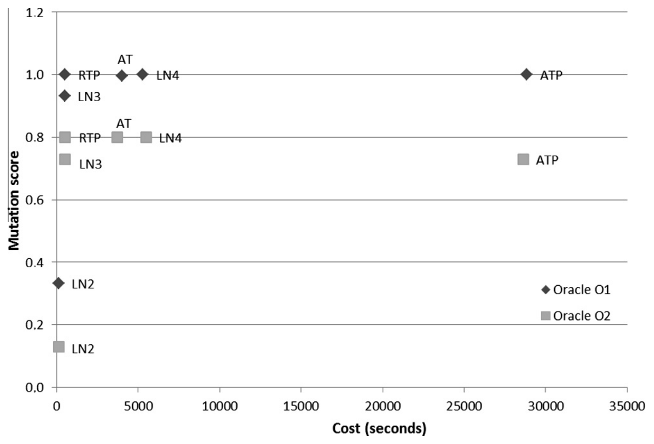

Fig. 14. Oracle O1 versus oracle O2 – mutation score versus preparation time (diamond = O1; square = O2).

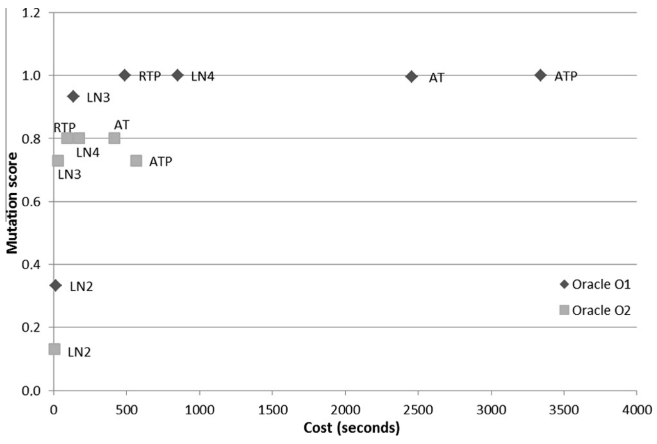

Fig. 15. Oracle O1 versus oracle O2 – mutation score versus execution time (diamond = O1; square = O2).

ommended; 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, while retaining an overall high mutation score when using oracle O1. Looking at the results for LN4, we see that there are significant differences in 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, the abstract model in combination with the better oracle (O1) actually performs better with respect to fault-detection when compared to the complete model combined with oracle O2. The latter also applies to LN2, LN3, RTP, and ATP.

The results for RTP when generating test suites from the complete model using the oracle O1 show a similar fault-detection level when 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 costs, however, differ. Notice that the cost of RTP is lower than that for AT, LN4, and ATP (complete model).

Finding 3: Reducing the level of detail in the test model significantly influenced cost effectiveness. The results show that both the costs and fault-detection ability are significantly lower for test suites generated from the abstract model, as compared to the complete model. However, when using the state-invariant oracle in addition to the state-pointer oracle, and either of the RTP, LN3, LN4, or ATP strategies, a comparable cost effectiveness could be obtained from the abstract test model as compared to test suites generated from the complete test model.

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

Sections 6.1, 6.2, 6.3 deal with conformance testing aimed at detecting deviations from specified system behavior when expected events were invoked on the SUT. We will now focus on a different type of testing, sneak-path testing. 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 behavior, in terms of extra transitions and actions.

Complementing conformance testing with sneak-path testing at an additional cost in preparation and execution time resulted in 11 mutants being killed – those 11 mutants were not killed by any of the six state-based coverage criteria. The execution of the sneak-path test suite on the abstract model killed 10 out of 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 when compared to the state-based coverage criteria investigated in this study. This demonstrates that these test strategies are complementary in order to catch different types of faults.

We are aware of the lack of formal statistical tests in this section. However, as the evidence found in the collected data is clear-cut and does not leave much place for uncertainty, we consider this as no threat to the drawn conclusions.

Finding 4: The results indicate quite strongly that sneak-path testing is a necessary step in state-based testing (SBT) 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.

7. Validity threats

This section discusses what the authors consider to be threats to validity.

A threat to internal validity is the fact that just one researcher was involved in the preparation and execution of the test cases. This was due to the lack of resources and a general lack of state-based testing (SBT) experience in the company. It is also worth noting that the purpose of the case study was to demonstrate the possible benefits ABB could gain by applying SBT in the future. The main purpose was not to study the interaction between the tester and the technology; the focus was directed at the actual achievements that could be obtained with respect to the cost and effectiveness via SBT. The industry needs help in introducing new techniques, and this is one pragmatic approach in order to demonstrate the possible advantages of this particular technique.

Another threat to internal validity could be related to fault seeding: both the generation of test cases and the insertion of faults to develop mutants were conducted by a researcher. The question that must be raised is: How can we ensure that the fault seeding was impartial and unbiased? The researcher could potentially influence the implementation of the test suites so as to be better at detecting certain types of faults. Ideally, the generation of test cases and fault seeding should be conducted by different

Table 19
Non-paired Wilcoxon signed-rank test comparing abstract (A) and complete (C) model – execution time.

H_0	Oracle	Measure	*p*-Value	$\widehat{A}_{12}$	Effect size	Result	Sign. diff. (CI)
A = C	O1	Exec. time	<2.2e−16	0.017	Large	A < C	Yes
A = C	O2	Exec. time	<2.2e−16	0.091	Large	A < C	Yes

Table 20
Non-paired Wilcoxon signed-rank test comparing abstract (A) and complete (C) model – mutation score.

H_0	Oracle	Measure	*p*-Value	$\widehat{A}_{12}$	Effect size	Result	Sign. diff. (CI)
A = C	O1	Mut. score	<2.2e−16	0	Large	A < C	Yes
A = C	O2	Mut. score	<2.2e−16	0.005	Large	A < C	Yes

Table 21
Non-paired Wilcoxon signed-rank test comparing abstract (A) and complete (C) model – all measures.

H_0	Measure	*p*-Value	$\widehat{A}_{12}$	Effect size	95% CI for $\widehat{A}_{12}$	Result	Sign. diff. (CI)
A = C	Prep. time	<2.2e−16	0.115	Large	[0.086, 0.152]	A < C	Yes
A = C	Exec. time	<2.2e−16	0.075	Large	[0.054, 0.104]	A < C	Yes
A = C	Mut. score	1.027e−10	0.170	Large	[0.132, 0.217]	A < C	Yes

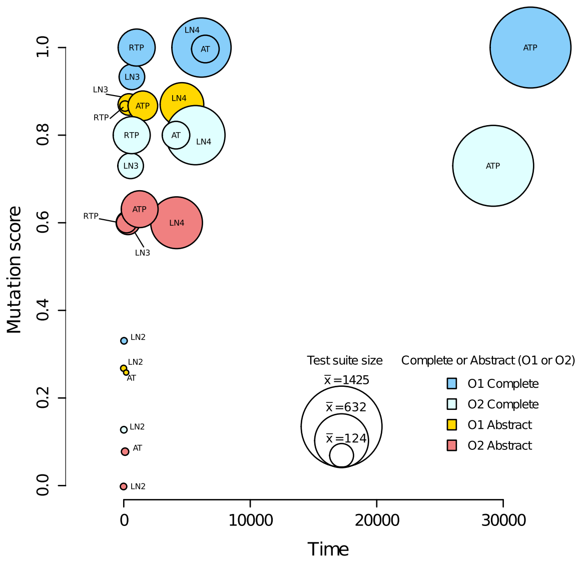

Fig. 16. Abstract versus complete test model – mutation score, test-suite size and time.

people. Nevertheless, since the test suites were automatically generated, following known algorithms, this is not considered to be a threat. As a result, the implementations of the algorithms do not suffer a great risk of being manipulated in favor of detecting specific faults. Furthermore, the seeded faults were actual errors introduced by engineers from ABB. Hence, we believe that, in this case, the researcher had no impact on how the faults were seeded.

One detected risk in terms of internal validity was the possible randomness in results for the RTP coverage criterion. This issue was handled by generating 30 random test trees, thus replicating the experiment for these criterion 30 times.

Even though the experiment was repeated 30 times, the number of seeded faults was low compared to mutation testing using artificial mutation operators. This may be a threat to the conclusion validity as the statistical power was low due to the small sample size. It is important to note, however, that this is a first study that should be complemented by additional studies with focus on increasing the sample size using mutation operators. Nevertheless, this study is unique for its use of actual industrial faults which positively affect the external validity and many key differences turned out to be statistically significant.

External validity covers 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 [56]. 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 actual faults when evaluating test strategies. The system that is the focus of this paper is highly representative of control systems with state-based behavior, thus improving the external validity. It is important to provide detailed context descriptions as we have done in this paper, such as system characteristics, development and testing procedures, so that others can relate the results to their own context. Moreover, in contrast to the majority of existing studies, which apply artificial faults, the faults used in this study are realistic faults collected in a field study conducted at ABB. In spite of these two factors, however, there are several issues that should be discussed.

First, let us consider 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. It is a highly appropriate and relevant case, as it was developed in an industrial context, and represents a real system, and is of real-world importance. The SUT is a typical example of control systems: It is a device that controls the movement of machines in industrial production by supervising inputs from a number of sensors. The characteristics of the selected system are expected to be similar for many control systems, which may increase the possibility of these results being generally valid for these types of systems.

In previous studies, where artificial mutation operators were applied to the evaluation of test strategies, external validity was considered uncertain. The use of actual faults when generating mutant programs is not common practice in testing research. In this study, only 26 mutants were applied, but the seeded faults were real and manually extracted from a field study as described in Section 3. But again, it is not certain that the results apply to other organizations because the types of faults that are introduced into a system depend on the engineers working on the system. Keeping the preparation and execution time in mind, the feasibility of the study would be threatened by a dramatic increase in the number of mutants. To avoid the masking of faults, only one fault was seeded per mutant program. Thus, like other studies, e.g. [38], this study only evaluates the detection of single faults. Complex fault patterns and interactions have not been accounted for. As with any empirical studies, however, this study should be replicated for other types of faults and other control systems in order to make the results more convincing.

Reliability concerns to what extent the data and the analysis are dependent on the specific researchers [56] meaning that unclear descriptions of data collection procedures may give different results in subsequent replications of the study. This is addressed by providing a detailed description of the study design and analysis.

8. Discussion

In this study, we have seen that sneak-path testing and the choice of coverage criterion, test oracle, and test model have a large impact on cost effectiveness.

By looking at the reported results on fault-detection ability, we see that ATP killed more mutants (100%) in this study when compared to results reported in [14,25,35]. The latter reported a mutation score as low as 54%. The relationship between the fault-detection ability of AT versus ATP presented in [25], however, seems to be rather consistent with the results in this study. In spite of the differences between the study of Briand et al. [12] and this study, our results support the findings of Briand et al.; with the exception of the findings showing that AT did not provide 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. However, the study of Briand et al. included many more mutants, albeit artificial. Our results showed that AT obtained a very low mutation score when applied to the abstract test model.

Empirical studies on RTP have, on the other hand, shown that, in terms of cost and effectiveness, this particular criterion is a compromise between the weak AT and the more expensive ATP criteria [12]. Our results support these findings. Note, however, that also for RTP, the reported results on fault detection in existing studies are highly variable (22–90%). We also found that LN2 is not recommended as a testing strategy due to its very low fault-detection ability, both when applied to the complete and abstract test models in our case study. LN3, on the other hand, obtained results comparable to RTP.

Although limited, existing research has suggested that the oracle used when testing has a large influence on fault-detection ability [13,57–59]. Overall, our study empirically supports these findings. For all strategies, the state-invariant oracle obtained significantly higher mutation scores than those that were obtained by the state-pointer oracle. AT, RTP, ATP and LN4 all provided the highest mutation scores when generated from the complete model used with the state-invariant oracle. Of these, RTP had the lowest costs. Using the state-pointer oracle, still based on the complete test model, the effectiveness was reduced: AT, RTP and LN4 killed 80% of mutants. Only the study of Briand et al. [13] is more precisely comparable as they also studied one of the coverage criteria in the focus of this study, namely RTP. Results for RTP indicated a 25% increase in fault-detection when using the strongest oracle. The study of Briand et al. differs from this study in the following manners: (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 fully precise oracle capturing exact expected results; 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 one of the researchers. (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. [13] found great variations in results, both studies suggest significant improvements in fault-detection when using a stronger oracle —yet at a significantly higher cost. For RTP, this applied to both preparation and execution time.

Reducing the test-suite size by removing details from the test model is yet another area of related work where no studies have been carried out within the context of SBT. Nevertheless, the desire of reducing test-suite sizes has received a significant amount of attention. There is a trade-off between a sufficiently detailed level in the test model and the cost effectiveness of the resulting test cases. The reduced costs of generating tests may, on the other hand, increase the number of undetected faults.

In our case, 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 described earlier, however, a large part of the generated test cases were infeasible due to guard conditions that could not be satisfied. The reason for this was that sub-state specific values could not be controlled in the same way as when those 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 an input for test generation proved to have acceptable fault-detection effectiveness combined with certain coverage criteria and the state-invariant oracle, we must take into account the omitted details and be aware of those parts that cannot be tested based on the model [60]. We found that the mutants that were seeded in the removed sub-states were not detected. An overall trend in the results for the mutation score was that the abstract test models obtained lower mutation scores than those that were achieved by the complete test models. Interestingly, we observed that by removing details from the test model and using a stronger criterion, the results revealed that a comparable cost effectiveness was obtained when compared to test suites generated from the complete test model. Although generated based on a different idea, our results thus support the findings of Wong et al. [27] that test-set minimization can greatly reduce the evaluation costs, and thus the cost of testing, with limited loss in fault-detection effectiveness. (This is, however, dependent on the choice of coverage criteria and oracle.)

The sneak-path test suite detected the eleven remaining mutants that were not killed by any of the conformance test suites. This demonstrates that sneak-path testing is complementary to coverage criteria in order to catch different types of faults. Our results support the recommendation of Binder [32] and the conclusions drawn in the study of Mouchawrab et al. [38]: Testing sneak paths is an essential component of SBT in practice. The additional cost is justified by a significant increase in fault-detection effectiveness, especially in a safety–critical context where robustness to unexpected events is often crucial.

This study contributes to existing research with its realistic context both in terms of the SUT and by using real faults when evaluating the testing strategies. The level of details in the test model is most often insufficiently specified; we have provided as detailed information as possible regarding the test models. The large majority of existing studies on cost effectiveness related to SBT, with the exception of [44], utilize small, non-industrial or example cases that are significantly smaller or less complex than the SUT in our study. For example, recall that the number of tests generated for ATP in this study was 1,425 as compared to 34 in [14,25]. In particular, recall the frequent use of the Cruise Control case in existing research. The number of seeded mutants is in most cases higher than in this study, but then again, the seeded faults are primarily artificial in existing studies. Nevertheless, the latter difference is interesting given the lack of studies on artificial versus real faults.

Our contributions also include comparing two different oracles. The applied oracle is most often not specified in existing research; the lack of oracle information makes it difficult to provide meaningful comparisons of the results. Only Briand et al. [13] specifically addressed and, in fact, compared the type of oracles used.

Existing research rarely reports the cost of SBT; in particular not when comparing several strategies. The focus has mostly been directed towards fault-detection effectiveness. The studies that report the cost of test strategies, primarily report the test-suite size. Our results contribute by also reporting the cost in terms of preparation and execution times. The measuring strategy presented in this paper is but one way to measure cost.

9. Conclusions

In this paper, we reported on an industrial case study that evaluated the cost effectiveness of state-based testing (SBT), i.e., model-based testing using state models, by studying the influence of four testing aspects: coverage criteria, test oracles, test models and sneak paths. Although SBT is not a new area of research, this paper was motivated by a lack of realism in studies evaluating SBT. Moreover, existing research has primarily focused on separate aspects of SBT, e.g., test oracles, test strategies. In this paper, however, we consider several aspects of SBT that may influence its cost effectiveness, and most importantly, evaluate these aspects together. Our overall goal was to achieve maximum realism and understand the interplay between selected test strategies, oracles, and modeling precision. The comparison is exclusively based on real faults.

In this study, we evaluated six SBT 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 results of studying the coverage criteria as applied to a complete test model and the state-invariant test oracle, indicate that LN2 may be too weak a testing strategy. The remaining testing strategies yield test suites that kill the seeded faults (except for sneak paths). When also considering the costs, significant differences were observed. ATP was the most expensive criterion. RTP appeared to be the most cost-effective strategy (closely followed by LN3).

By using the less rigorous state-pointer oracle, only minor differences in preparation time were observed as compared to using the state-invariant oracle. Execution time, on the other hand, was significantly lower when applying the former oracle for all six strategies. However, effectiveness was negatively impacted. Yet, 80% of the mutants (except sneak paths) were killed by AT, RTP, and LN4.

In addition, reducing the level of detail in the test model resulted both in lower costs and effectiveness as compared to the complete model. As we can see from Fig. 16, however, the results for LN3, RTP, LN4 and ATP show that the obtained mutation score exceeds 80% even when using the less detailed test model.

Complementing conformance testing with sneak-path testing resulted in killing all the remaining mutants, though at an additional but reasonable cost in preparation and execution times. This demonstrates that these test strategies complement each other 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 (SBT) 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.

Each of the testing aspects, coverage criteria, test oracles, and test models influence cost effectiveness, and constitute a tradeoff between increasing fault-detection effectiveness and reducing costs. These aspects must be carefully considered when selecting a strategy. However, regardless of these choices, sneak-path testing is a necessary step in SBT as the presence of sneak paths is undetectable by conformance testing. The results in this study regarding cost and effectiveness of SBT should provide useful guidance for industry on how to select appropriate testing strategies by accounting for their relative cost and the criticality of the SUT.

Acknowledgements

Nina E. Holt was funded by Simula Research Laboratory and The Research Council of Norway through the industry project EVISOFT (EVidence based Improvement of SOFTware engineering). Lionel Briand has been supported by the National Research Fund, Luxembourg (FNR/P10/03). Richard Torkar was partly funded by The Knowledge Foundation (KKS) through the project 20130085 Testing of Critical System Characteristics (TOCSYC).

References

[1] M. Utting, B. Legeard, Practical Model-Based Testing: A Tools Approach, Morgan-Kaufmann, 2006.
[2] T. Chow, Testing software design modeled by finite-state machines, IEEE Trans. Softw. Eng. 4 (3) (1978) 178–187.
[3] L. Briand, 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, J. Whittaker, Model-based software testing, in: J.J. Marciniak (Ed.), Encyclopedia of Software Engineering, 2002.
[5] S. Ali, L. Briand, M. Rehman, H. Asghar, M. Iqbal, A. Nadeem, A state-based approach to integration testing based on UML models, Inf. Softw. Technol. 49 (2007) 1087–1106.
[6] A. Dias Neto, R. Subramanyan, M. Vieira, 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, first ed., Newnes, 2006.
[8] M. Vieira, X. Song, G. Matos, S. Storck, R. Tanikella, 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. <http://www.elvior.com/motes/d-mint> (accessed 10.13).
[10] J. Feldstein, Model-Based Testing using IBM Rational Functional Tester, Developer Works, IBM, 2005.
[11] Y. Gurevich, W. Schulte, N. Tillmann, M. Veanes, Model-Based Testing with SpecExplorer, Microsoft Research, 2009.
[12] L. Briand, Y. Labiche, Y. Wang, Using simulation to empirically investigate test coverage criteria based on statechart, in: Proceedings of the 26th International Conference on Software Engineering, 2004.
[13] L. Briand, M. Di Penta, Y. Labiche, Assessing and improving state-based class testing: A series of experiments, IEEE Trans. Softw. Eng. 30 (11) (2004) 770–793.
[14] J. Offutt, A. Abdurazik, Generating tests from UML specifications, in: Proceedings of the 2nd International Conference on the Unified Modeling, Language, 1999.
[15] MOTES. <http://www.elvior.com/motes/generator. (accessed 10.13).
[16] A. Hartman, K. Nagin, The AGEDIS tools for model based testing, in: International Symposium on Software Testing and Analysis (ISSTA '04), 2004.
[17] Working with TestConductor and Automatic Test Generation (ATG), IBM. <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 07.12).
[18] T. Santen, D. Seifert, TEAGER – test automation for UML state machines, in: Proceeding of Software Engineering, Leipzig, 2006.
[19] Conformiq Tool Suite. <http://www.conformiq.com/> (accessed 10.13).
[20] P. Black, V. Okun, Y. Yesha, Mutation operators for specifications, in: ASE'2000, 15th Automated Software Engineering Conference, Grenoble, France, 2000.
[21] A. Offutt, A. Lee, G. Rothermel, R. Untch, C. Zapf, An experimental determination of sufficient mutation operators, ACM Trans. Softw. Eng. Methodol. 5 (2) (1996) 99–118.
[22] J. Andrews, L. Briand, Y. Labiche, Is mutation an appropriate tool for testing experiments? in: ICSE '05: Proceedings of the 27th International Conference on Software Engineering, 2005.
[23] M. Thévenod-Fosse, P. Daran, Software error analysis: a real case study involving real faults and mutations, in: Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis, 1996.

[24] J. Andrews, L. Briand, Y. Labiche, A. Namin, Using mutation analysis for assessing and comparing testing coverage criteria, IEEE Trans. Softw. Eng. 32 (8) (2006) 608–624.
[25] J. Offutt, S. Liu, A. Abdurazik, P. Ammann, Generating test data from state-based specifications, Softw. Test. Verif. Reliab. 13 (1) (2003) 25–53.
[26] M. Heimdahl, 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.
[27] W. Wong, J. Horgan, S. London, 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, 1995.
[28] S. Ali, H. Hemmati, N. Holt, E. Arisholm, L. Briand, Model Transformations as a Strategy to Automate Model-Based Testing: A Tool and Industrial Case Studies, Simula Research Laboratory, 2010.
[29] R. Yin, Case Study Research Design and Methods, fourth ed., SAGE Publications, 2009.
[30] T. Pender, The UML Bible, Wiley, 2003.
[31] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
[32] R. Binder, Testing Object-Oriented Systems, Addison-Wesley, 2000.
[33] A. Offutt, Y. Xiong, S. Liu, Criteria for generating specification-based tests, in: Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99), 1999.
[34] Y.-S. Ma, J. Offutt, Y. Kwon, MuJava: an automated class mutation system, Softw. Test. Verif. Reliab. 15 (2) (2005) 97–133.
[35] A. Paradkar, Plannable test selection criteria for FSMs extracted from operational specifications, in: Proceedings of the International Symposium On Software Reliability Eng. '2004, 2004.
[36] G. Antoniol, L. Briand, M. Di Penta, Y. Labiche, A case study using the round-trip strategy for state-based class testing, in: Proceedings of the 13th International Symposium on Software, Reliability Engineering (ISSRE'02), 2002.
[37] S. Mouchawrab, L. Briand, 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.
[38] S. Mouchawrab, L. Briand, Y. Labiche, M. Di Penta, Assessing, comparing, and combining state machine-based testing and structural testing: a series of experiments, IEEE Trans. Softw. Eng. 37 (2) (2011) 161–187.
[39] L. Briand, Y. Labiche, 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.
[40] L. Briand, Y. Labiche, Q. Lin, Improving the coverage criteria of UML state machines using data flow analysis, Softw. Test. Verif. Reliab. 20 (3) (2010) 177–207.
[41] H. Gomaa, Designing Concurrent, Distributed, and Real-Time Applications with UML, Addison-Wesley, 2000.
[42] A. Abdurazik, P. Ammann, W. Ding, J. Offutt, Evaluation of three specification-based testing criteria, in: Proceedings of the 6th IEEE International Conference on Complex Computer Systems, 2000.
[43] K. Bogdanov, M. Holcombe, Statechart testing method for aircraft control systems, Softw. Test. Verif. Reliab. 11 (1) (2001) 39–54.
[44] P. Chevalley, 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.
[45] N. Holt, R. Torkar, L. Briand, K. Hansen, State-based testing: industrial evaluation of the cost-effectiveness of round-trip path and sneak-path strategies, in: IEEE International Symposium on Software Reliability Engineering, 2012.
[46] Kermeta – Breathe Life into Your Metamodels. <http://www.kermeta.org/> (accessed 10.13).
[47] N. Holt, E. Arisholm, 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.
[48] N.E. Holt, B. Anda, K. Asskildt, L.C. Briand, J. Endresen, S. Frøystein, Experiences with precise state modeling in an industrial safety critical system, in: Critical Systems Development Using Modeling Languages, CSDUML'06, Genova, 2006.
[49] ATLAS Transformation Language (ATL). <http://www.eclipse.org/atl/> (accessed 10.13).
[50] MOFScript Model Transformation Tool. <http://marketplace.eclipse.org/content/mofscript-model-transformation-tool#.UktD3tJHKPw> (accessed 10.13).
[51] R. DeMillo, A. Offutt, Constraint-based automatic test data generation, IEEE Trans. Softw. Eng. 17 (9) (1991) 900–910.
[52] P. McMinn, Search-based software test data generation: a survey, Softw. Test. Verif. Reliab. 14 (2) (2004) 105–156.
[53] M. Alshraideh, L. Bottaci, Search-based software test data generation for string data using program-specific search operators, Softw. Test. Verif. Reliab. 16 (3) (2006) 175–203.
[54] R. Wilcox, Fundamentals of Modern Statistical Methods: Substantially Improving Power and Accuracy, Springer Verlag, 2001.
[55] A. Vargha, H. Delaney, A critique and improvement of the CL common language effect size statistics of McGraw and Wong, J. Educ. Behav. Stat. 25 (2) (2000) 101–132.
[56] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empiric. Softw. Eng. 14 (2) (2009) 131–164.
[57] Q. Xie, A. Memon, Designing and comparing automated test oracles for GUI-based software applications, ACM Trans. Softw. Eng. Methodol. 16 (1) (2007).
[58] M. Staats, M. Whalen, M. Heimdahl, Better testing through oracle selection, in: Proceeding of the 33rd International Conference on Software Engineering, 2011.
[59] M. Staats, M. Whalen, M. Heimdahl, Programs, tests, and oracles: the foundations of testing revisited, in: Proceeding of the 33rd International Conference on Software Engineering, 2011.
[60] M. Utting, A. Pretschner, B. Legeard, A Taxonomy of Model-Based Testing, Working Paper Series, University of Waikato, Department of Computer Science, No. 04/2006, 2006.