State-Based Testing: Industrial Evaluation of the Cost-Effectiveness of Round-Trip Path and Sneak-Path Strategies

Nina Elisabeth Holt[1], Richard Torkar[2], Lionel Briand[1,3], and Kai Hansen[4]

1 Simula Research Laboratory
Lysaker, Norway
e-mail: ninaeho@simula.no

2 Division of Software Engineering, Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
e-mail: richard.torkar@chalmers.se

3 SnT Centre, University of Luxembourg
Luxembourg
e-mail: lionel.briand@uni.lu

4 ABB
Billingstad, Norway
e-mail: kai.hansen@no.abb.com

Abstract— **In the context of safety-critical software development, one important step in ensuring safe behavior is conformance testing, i.e., checking compliance between expected behavior and implementation. Round-trip path testing (RTP) is one example of conformance testing. Another essential step, however, is sneak-path testing, that is testing of how software reacts to unexpected events for a particular system state. Despite the importance of being systematic while testing, all testing activities take place, even for safety-critical software, under resource constraints. In this paper, we present an empirical evaluation of the cost-effectiveness of RTP when combined with sneak-path testing in the context of an industrial control system. Results highlight the importance of sneak-path testing since unexpected behavior is shown to be difficult to detect by other common, state-based test strategies. Results also suggest that sneak-path testing is a cost-effective supplement to RTP.**

Round-trip path testing; sneak-path testing; UML; state-based testing; cost-effectiveness; automated tool support; empirical evaluation; industrial case study

I. INTRODUCTION

Software testing is often conducted as a manual, ad hoc task, as compared to following an automated and more systematic procedure. Consequently, testing is likely to be incomplete and costly to ensure the required level of dependability. Safety-critical software systems must be tested so as to ensure its safe behavior. In order for industry to make the right choices when deciding on how to test their software, more knowledge must be gained about how various test strategies compare in terms of cost-effectiveness. As thorough software testing is an expensive task, reducing the cost of testing while ensuring sufficient fault-detection effectiveness is of common interest to industry.

When using automated testing to check the compliance of implementations against their specifications, model-based testing has become a popular area of research and practice. Test models, for example expressed as UML state machines, describe the expected behavior of the software and provide the basis for systematic and automated generation of test suites. In contrast to conformance testing, which aims at detecting deviations from specified system behavior, when expected inputs are given to the system under test (SUT), sneak-path testing [1] feeds the SUT with unexpected events that should not trigger any change of state in the SUT. For each state in the SUT, all possible events not specified for that state are invoked. This technique is intended to catch faults that introduce undesired, additional behavior, in terms of extra transitions and actions.

This paper assesses round-trip path (RTP) [1] and sneak-path testing in terms of cost and fault-detection ability in an industrial control system. Though existing work has already addressed this problem, the current paper goes much further in terms of obtaining realistic results. The evaluation was carried out in the context of a safety-monitoring component in a control system that was developed by ABB using UML state machines [2] and implemented according to the extended state-design pattern [3]. We used the TRUST tool [4] to automatically generate the RTP and the sneak-path test suites. Furthermore, the two test strategies were evaluated in terms of cost and effectiveness based on real faults collected in a field study at ABB. Our overall goal was to achieve maximum realism.

The remainder of this paper is structured as follows. Section II reports on related work. The case-study design is described in Section III, whereas the obtained results are presented in Section IV. Finally, Section V concludes the paper.

II. BACKGROUND AND RELATED WORK

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

As one of the first studies on deriving tests from finite-state machines (FSM), Chow presented a testing strategy known as the *W*-method [5]. Chow based the method on test sequences derived from a spanning tree generated from the FSM describing the SUT behavior. The method was modified by Binder [1] to be used in a UML context and referred to as round-trip path (RTP) testing. The *W*-method and Binder's adaptation traverse the transition tree to cover all paths, stopping a path traversal when reaching a state that was already reached in a previous path. As opposed to Chow's method, Binder assumes that it is possible to directly check the state invariant rather than relying on an identification sequence. Binder denotes the set of paths covered by the tree as RTPs as they capture all transition sequences that begin and end with the same state (with no repetitions of states other than the sequence start and end state) and simple paths from the initial to the final state of the statechart. A prerequisite to use Binder's approach is to flatten the system statecharts [1, 6], i.e., all hierarchy and concurrency [7] must be removed.

Whereas test strategies like RTP seek to compare explicitly the modeled behavior to the actual software execution, it is also important to test whether or not the software handles unspecified behavior, also called sneak paths [1]. State machines are usually incompletely specified and this is normally interpreted as events on which the system should not react, i.e., changing states or performing actions. Sneak-path testing exercises every unspecified event in all states. In other words, sneak-path testing aims to verify the absence of unintentional sneak paths in the software under test as they may have catastrophic consequences in safety critical systems. As defined by Binder [1], "a sneak path is possible for each unspecified transition and for all situations in which the predicate of a guarded transition evaluates as FALSE".

One common 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 [6]. Mutants are identified through the static analysis of the source code by a mutation system [9]. 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 total number of mutants, is referred to as the mutation score and is used as a measure of a test suite's fault-detection effectiveness. Some mutants may be functionally equivalent to the correct version of the SUT. These are called *equivalent mutants* and should not be included in the pool of mutants used for analysis. We will follow the same procedure in our study, except for the important fact that mutants will be based on actual faults collected in the field.

Motivated by the lack of extensible and configurable model-based testing tools, Ali *et al.* [4] proposed a MBT tool, TRansformation-based tool for Uml-baSed Testing (TRUST), whose software architecture and implementation strategy facilitated its customization to different contexts by supporting configurable and extensible features such as input models, test models, coverage criteria, test data generation strategies, and test script languages. TRUST was applied in our study. The state machine flattening component of TRUST was implemented using Kermeta; a metaprogramming environment based on an object-oriented domain specific language optimized for metamodel engineering [10]. Moreover, Kermeta was also used to remove infeasible state combinations and transitions from flattened state machines.

Investigating the impact of RTP testing on cost and fault detection when compared to structural testing, Mouchawrab *et al.* [11] conducted a series of controlled experiments. The study was a replication of [12] where one of the findings was that not testing self-transitions [2] 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 [1]. Results showed that sneak-path testing clearly improved fault detection and, thus, strongly suggests complementing RTP with sneak-path testing. It is not, however, recommended to include explicit self-transitions in state models as they may become too complex. No other empirical study evaluates the testing of sneak paths and, more importantly, there are no studies in realistic industrial contexts.

III. CASE STUDY DESIGN

This section describes how our empirical evaluation of round-trip path (RTP) and sneak-path testing was carried out. The guidelines of Runeson and Höst [13] were followed when designing the case study.

A. Research Question

In this study, we aimed to investigate the cost-effectiveness when combining RTP testing with sneak-path testing. Based on this objective, we seek to answer the following research questions:

1. What is the cost-effectiveness of RTP and sneak-path testing?
2. How does complementing sneak-path testing with RTP testing influence the cost and fault detection rates?

B. Case and Subject Selection

The case study was conducted in the context of a software process improvement project at ABB. A primary business area concerns power and automation technologies for which ABB develops software and hardware. The project was initiated by the company in cooperation with Simula Research Laboratory to investigate which UML diagrams may be beneficial to ABB in the process of going from the specification to the implementation phase, and for improving testing procedures. It

provided a unique opportunity to assess the usage of precise, statechart-driven UML modeling and to evaluate state-based testing (SBT) in a realistic safety-critical development environment.

In order to satisfy safety standards (EN 954 and IEC 61508) and enhance the safety-critical behavior of their industrial machines, ABB developed a new version of a safety-critical system, the safety board, for supervising industrial machines. The safety board can safeguard up to six robots by itself and can be interconnected to many more via a programmable logic controller (PLC). It was implemented on a hardware redundant computer in order to achieve the required safety integrity level (SIL). The focus of this study is a 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 the components requests a stop or if a fault is detected. It interfaces with an optional safety bus to enable a remote stop and a reliable exchange of system status information. This sub system was chosen for this study as it exhibits a complex state-based behavior that can be modeled as a UML state machine. Complemented by constraints specifying state invariants, the state machine was the main source in the process of deriving automated test oracles. The selected module was a typical control system, which controls the physical movement of industrial machines by supervising inputs from a number of sensors.

C. Data Collection Procedures

To measure the cost-effectiveness of SBT, four surrogate measures were used, consistent with the study of Briand *et al.* [12] and Briand and Labiche [14].

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.

A Technical Requirements Specification, developed by the business unit in cooperation with the scientists from NOCRC, was the starting point for developing a common understanding of the system. The modeling and implementation was a cooperative activity between Simula and ABB. The testing, however, was exclusively performed by one of the researchers from Simula.

The remainder of this section addresses the data collection procedures. The main activities include: 1) Preparation of test models. 2) Collecting fault data for creating mutants. 3) Extending and configuring the testing tool TRUST. 4) Generation of test suites. 5) Execution of test suites on initial and mutated programs.

1) Preparation of Test Models: As part of evaluating a number of SBT strategies, a model-based testing tool TRUST [4] was developed and used to automatically generate test suites. The model was tested using TRUST when configured with the RTP criterion. However, some adjustments had to be done before being used as input to TRUST. As the flattening component does not support transitions that cross state borders, those transitions had to be re-modeled to and from the super-state edges. This also implied adding initial states, entry points, and exit points in several of the super states. The state-machine flattener does not support multiple events on a single transition. Such transitions were thus transformed to singe-event transitions. Note that the applied changes did not affect the functionality as originally implemented.

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 I and the flattened state machine described in Table II as 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 [4].

Executing the generated test suites showed that a large number of test cases failed. Analyses of the execution results showed that this was due to infeasible test cases as a result of the selected test data values on guarded transitions. The MOFScript transformation was modified to provide the required test data.

Furthermore, even when a system is carefully designed and implemented, there is always a 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 experimentation.

TABLE I. CONFIGURATION PARAMETERS OF TRUST

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

TABLE II. FEATURES SUMMARY OF THE HIERARCHICAL SCALE OF STATE MACHINES – 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 III. FEATURES SUMMARY OF THE HIERARCHICAL SCALE OF STATE MACHINES – 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	61(17)	349 (107)

To enable a realistic evaluation of SBT strategies, a field study was planned for the purpose of collecting fault data to be used in the generation of mutants. For this reason, the original version of the system was modified to include a fourth operation mode—the *ExtraSlow* mode. 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, six simple-composite states, and a maximum hierarchy level of two. The unflattened state machine contains 61 transitions. Having both concurrent and hierarchical states, the state machine had to be flattened before being used as input to test case generation. The state-machine flattening component of TRUST was used for this purpose. In the outset, the flattened model consisted of 82 states and 508 transitions, of which 193 were guarded.

However, as addressed above, the flattened model contained infeasible state combinations and transitions, as the current version of the state-machine flattener does not remove these automatically. The user can specify preferences in a provided Kermeta transformation. The outcome of the transformation is a model where these combinations are excluded. After removing infeasible transitions, both due to incompatible state invariants (12 state combination, 145 transitions) and guards that will never become true (14 transitions), the state machine included 68 simple states (excluding initial and final state) and 349 transitions, of which 107 are guarded. The characteristics of the unflattened and flattened UML state machines are summarized in Table III.

The resulting code for the SUT consisted of 26 classes and 3,372 LOC (1,227 h, 2,145 cpp) (without blank lines).

2) Collecting Fault Data for Creating Mutants: For the purpose of evaluating the cost-effectiveness of SBT strategies and to increase the external validity of the results, fault data was collected in a global field study to generate realistic, mutated versions of the SUT. The field study was conducted at ABB's departments in Västerås, Baden and Shanghai. Having different UML and domain knowledge, ten ABB engineers were asked to implement a change task to the SUT. The majority of the subjects had more than three years experience in using UML. Many of the subjects had between one and three years experience within the domain. Eight of the subjects were experienced in C++.

They were instructed to modify both the model and code. Participants were strictly asked to work individually. One researcher supervised the sessions. The maintenance task itself was initially suggested at a high level by ABB; however, 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 implementing the maintenance task, was developed by one researcher and exhaustively tested with each of the coverage criteria in focus of this study. The faults extracted from the field-experiment data were manually inserted into this correctly modified version of the SUT to produce mutant versions.

Manual code inspections of the ten solutions were used to collect actual faults. In total, 26 faults were detected from model and code inspections. These faults varied from occurring in both model and code to only occurring in the code. Note that because the objective was to compare the fault-detection abilities of testing criteria, it was crucial for the seeded faults not to 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 of the code by manually seeding one fault per program. The faults from the code reflect the following modeling errors:

- **Missing transitions (Figure 1):** 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.

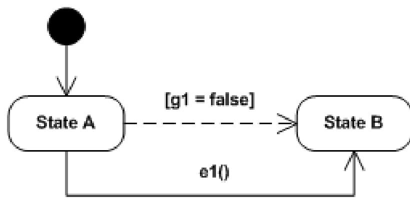

Figure 1. Missing Guarded Transition

- **Additional transitions (Figure 2):** 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.

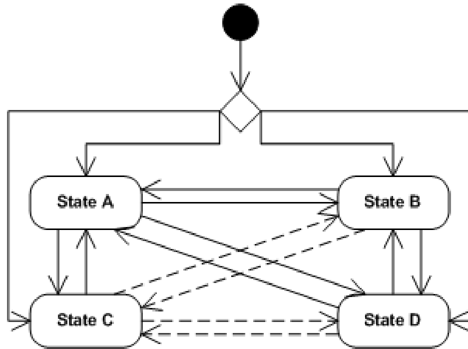

Figure 2. Additional Transitions to and from `State C`

- **Guards that were not correctly updated (Figure 3):** 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.

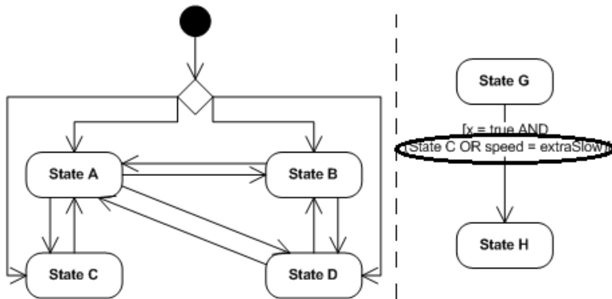

Figure 3. Incorrect Guard on Transition from `State G` to `State H`

- **Guards that were modified which should not have been changed (Figure 4):** 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`.

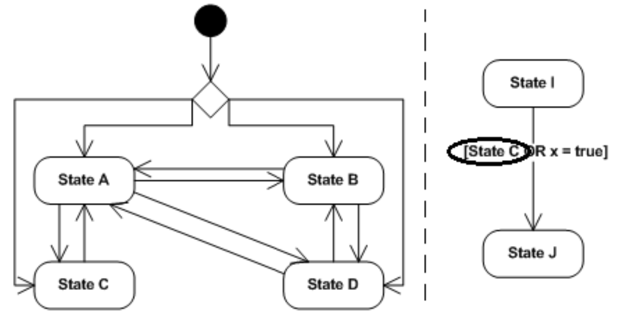

Figure 4. Incorrect Guard on Transition from `State I` to `State J`

- **Erroneous on-entry behavior (Figure 5):** 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.

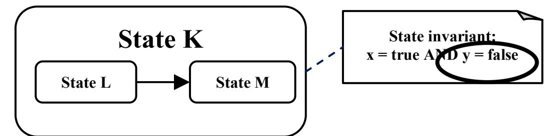

Figure 5. Erroneous On-Entry Behavior that Set State Variables

- **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 faults of this type, the model should reflect the preferred behavior of the system when exposed to unexpected events. At this stage, our model had no support for the handling of such unexpected behavior. Hence, as sneak paths could not be caught by any conformance test suite generated from the model, only 15 out of 26 mutant programs were detectable by the conformance (RTP) test suites.

It is important to note that the test strategies were selected before collecting fault data and that these faults were introduced in a well designed object-oriented system (relying on the extended state pattern).

3) Extending and Configuring the Testing Tool TRUST: In this study, we configured TRUST to generate RTP test suites using the state-invariant oracle. RTP was implemented following the breadth-first algorithm. However, as TRUST only supported the weaker variant of the RTP criterion (i.e., only one OR clause is tested on guarded transitions), the state machine itself was modified so that guarded transitions with OR clauses were split in one transition per clause so as to ensure that each clause was tested.

Sneak-path testing can be implemented in several ways. In our case study, a Kermeta [10] program was created for the

purpose of generating separate state machines for each state of the SUT; 68 state machines in total. The state machines included the initialization state, a path from the initialization state that would lead to the state under test, and the state under test itself. The complete transformation took less than an hour. TRUST was then configured with the all-transitions coverage criterion (AT) to generate test trees. MOFScript was, as for the previously generated test suites, used to create the concrete test cases. Thus, by traversing all paths in the test tree for each state machine, the sneak-path test suite was generated. As for RTP, the state-invariant oracle was applied.

A test procedure, as specified by Binder [1], was followed, which includes the following steps:

1. Place the SUT in the corresponding state.
2. Apply the illegal event by sending a message or forcing the virtual machine to generate the desired event.
3. Check that the actual response matches the specified response. In our case, this means that no transition takes place, and no action or effect is executed.
4. Check that the resultant state is unchanged.

4) Generation of Test Suites: TRUST was used to automatically generate executable test suites from the test model previously introduced in Section 1. The following steps summarize the automated experimental process that was followed:

1. Flatten the input state machine.
2. Select the test adequacy criterion.
3. Construct abstract test cases in the form of a test tree. The algorithms used to traverse the state machine are described above.
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. Ensure that test results reveal no errors. Then build and execute the test suite on each of the mutant programs.
7. Analyze test results provided by the state-invariant oracle on all mutants.

The prepared test model was used as input model to TRUST. As the state-based criteria are defined for finite state machines, a prerequisite for generating the test suites is to use an input state machine without concurrency and hierarchy. The first step in TRUST was thus to flatten [6] the test model, i.e., removing hierarchy and concurrency from the model as previously described. To create the abstract test cases, in the shape of a test tree, each of the algorithms for obtaining test suites satisfying the coverage criteria under study, were applied on the flattened state machine. TRUST then created concrete test cases using MOFScript, which took the flattened state machine in addition to the generated test tree as input. In MOFScript, the test tree was traversed path by path, i.e., each path produced one test case. A separate C++ file was created for each test case. A main C++ file was generated where each of the test cases were invoked. Each test case was invoked with a new object of the SUT.

The test suites were 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. When the test suites executed successfully, the test suites were run on the mutant versions of the SUT.

For the RTP coverage criterion, the generation of test trees from state machines is not deterministic as several test trees can possibly satisfy the criterion. The explanation for this is that the structure of the tree depends on the sequence of the selected outgoing transitions when traversing the state machine. However, due to possible differences in the fault-detection level of the various test suites, the results of executing different test suites that fulfill the same test criterion may differ. Such random variations in the results were accounted for by repeating the experiment 30 times; 30 different trees were created using random selection of the order of outgoing transitions from states to generate distinct test suites. Being selected without replacement from the population of all possible trees that achieves each of the criteria, only trees distinct from the selected trees were added to the selection. The test suites were obtained by traversing each of the 30 test trees covering all paths.

The aim of sneak-path testing was to show that the mutants actually could be killed, not to show how many of the test cases that could kill each of the mutants. Therefore, it was decided to prepare and execute only the minimum number of sneak-path test cases that were required in order to kill all mutants.

5) Executing Test Suites: A batch file was created for each of the RTP and sneak-path test strategies, to automate the build, execution, and timing 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 file in the format of a text file was created for each test strategy. The result file contains the results for the correct version and the mutants. As illustrated in Figure 6, the RTP criterion was 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 by conformance test suites like RTP.

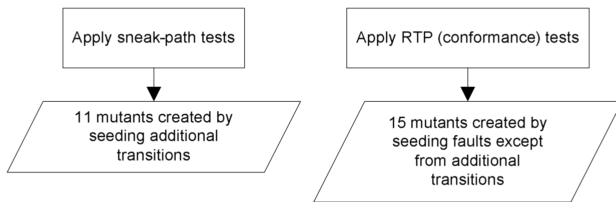

Figure 6. Overview of Test Execution on Mutants

D. Analysis Procedures

The analysis procedures are based on quantitative data that was used to explore the cost-effectiveness of RTP and sneak-path testing. JMP 7 [15] and Excel 2007 were used for data analysis.

The main features of the collected data, like central tendency, statistical variability, and distribution shape, were described using descriptive statistics. The summary statistics (the five-number summary) and the associated box plot were used for this purpose. The cost-effectiveness of the RTP and the sneak-path testing strategies were compared by exploring the collected data. Recall that cost-effectiveness was represented by the surrogate measures test suite size, preparation and execution time, and mutation score. Potential random variations in the mutation score across test suites for the RTP coverage criterion created a need for analyzing the distribution of the mutation scores. Through this analysis, we could present an overview of the expected percentage of mutants that are killed by the test strategies and at which cost.

E. Validity Procedures

This section discusses what the authors of this paper consider as threats to the validity.

A threat to internal validity is the fact that one researcher was involved in the preparation and execution of test cases. The rationale for this being lack of resources and a general lack of state-based testing (SBT) experience in the company. Also important to notice is that the purpose of the case study was to demonstrate the possible benefits ABB could gain by applying SBT in the future. That is, the main purpose was not to study the interaction between the tester and the technology; the focus was directed towards the actual achievements that could be obtained with respect to cost and effectiveness by introducing RTP and sneak-path testing. The industry needs help in introducing new techniques, and this is one pragmatic approach in order to demonstrate possible advantages of this particular technique.

Another threat to internal validity could be related to the 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 be ensured that the fault seeding was impartial and unbiased? The researcher could potentially influence the implementation of the test suites as to be better at detecting certain types of faults. Ideally, generation of test cases and fault seeding should be conducted by different people. Nevertheless, as the test suites were automatically generated, 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 claim that the researcher, in this case, had no impact on how the faults were seeded.

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

External validity concerns to what extent it is possible to generalize the findings, and to what extent the findings are of interest to other people outside the investigated case [13]. 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 the test strategies. The system in 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, like system characteristic, development and testing procedures, such 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 real 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 of all, 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. Even though it is a highly appropriate and relevant case, as it was developed grouping an industrial context, it is large, it represents a real system, and is of real world importance, drawing general conclusions is not possible from this one case. However, 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 valid for these types of systems.

In previous studies where artificial mutation operators have been applied in the evaluation of test strategies, the question of its impact on external validity has been raised. The use of actual faults when generating mutant programs is not common practice in testing research. In this study, although only 26 mutants were applied, the seeded faults were real and manually extracted from a field study as described in Section III. But again, it is not certain that the results apply to other organizations as it depends on the engineer what types of faults are introduced to a system. Hence, further studies are necessary to increase the external validity of the results. Having the preparation and execution time in mind, the feasibility of the study would be threatened by a dramatic increase in number of mutants. To avoid masking of faults, only one fault was seeded per mutant program. Thus, like other studies, e.g. [11], this study only evaluates the detection of single faults. Complex fault patterns and interactions have not been accounted for.

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

Reliability concerns to what extent the data and the analysis are dependent on the specific researchers [13] e.g., unclear descriptions of data collection procedures such that later replications of the study could give different results. This is addressed by providing a detailed description of the study design and analysis.

IV. RESULTS AND ANALYSIS

We now present the obtained results on cost and effectiveness for the round-trip path (RTP) criterion, and then the results for sneak-path testing to understand how sneak-path testing affects the results obtained with RTP.

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 was measured in seconds.

A. Round-Trip Path Testing

1) Cost: This section presents main features of the collected data on cost; namely test suite size, and preparation and execution time. First, however, please recall from Section III that in order to reduce the uncertainty in the mean, 30 trees were selected for RTP. Also note that the collected data on execution time does not include the 11 mutants that can only be detected by sneak paths; conformance test suites like RTP are not designed to detect sneak paths. The RTP test suites were only executed on the 15 mutants that could be killed by conformance test suites.

Configuring TRUST with the RTP criterion resulted in an average test suite size of 299. Descriptive statistics for preparation and execution time are provided in Table IV. The statistics include the minimum, the 25 percent quartile, the mean, the median, the 75 percent quartile, the maximum, and the standard deviation in the collected data for each coverage criterion.

Looking at the mean values, we observe that RTP used 531 seconds on preparing the test suite and 489 seconds on executing the test suite. The data in Table IV illustrate the variations in results among the generated trees.

The histogram and box plot in Figure 7 show the distribution of the time spent to prepare the test suites. As we can see, the histogram does not show a normal distribution; the distribution is skewed to the left. The minimum value was 484 and the maximum observed value was 607 seconds.

Also Figure 8, which displays the spread in the collected execution-time data, presents non-normally distributed observations; the RTP distribution is slightly skewed to the left. Looking at Table IV, we see that the minimum value was 341 seconds, the mean was 489 seconds, and the maximum value was 607 seconds.

TABLE IV. DESCRIPTIVE STATISTICS – PREPARATION AND EXECUTION TIME FOR THE ROUND-TRIP PATH TEST SUITE

	Min	***Q1***	***Mean***	***Median***	***Q3***	***Max***	***St Dev***	***n***
Prep	484	512	531	525	545	607	28	30
Exec	341	460	489	504	524	607	54	30

TABLE V. INTER QUARTILE RANGES – PREPARATION AND EXECUTION TIME

	Q1	***Q3***	***Q3-Q1***	***IQR***	***Lower limit***	***Upper limit***
Prep	512	545	33	50	462	595
Exec	460	524	64	96	364	621

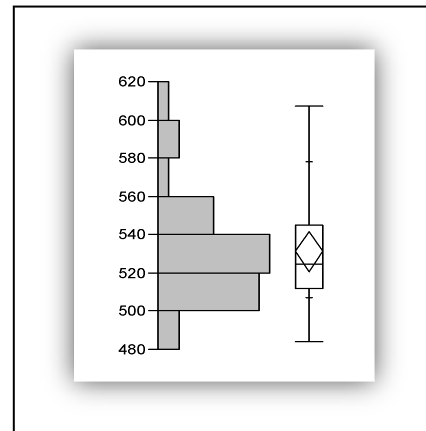

Figure 7. Distribution – Preparation Time

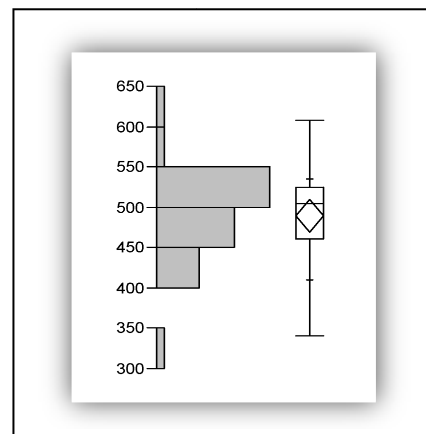

Figure 8. Distribution – Execution Time

2) Effectiveness: The obtained results on mutation score, the surrogate measure of effectiveness, show that all 30 RTP test suites killed each of the 15 mutants that could be killed by

the conformance test suite provided by RTP. Thus, the mean mutation score was in this case 1.

B. Sneak-Path Testing

1) Cost: The size of the sneak-path test suite is equal to the number of states in the SUT (68). The length of each test case depends on two factors: 1) The length of the path that must be traversed in order to reach the particular state to be tested. 2) The number of known unexpected events for the state.

Recall that 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. Also note that each test case was prepared and executed separately.

Table VI shows the time spent on preparing the sneak-path test cases. The collected data shows that TRUST used 305 seconds on preparing the sneak-path test cases. The time varies from 23 seconds to 34 seconds. A realistic estimate for the complete sneak-path test suite is 1,885 seconds ((305 ÷ 11) × 68) in preparation time.

Table VI also displays the execution time observed. The minimum and maximum times were, respectively, three and seven seconds. TRUST used 51 seconds to execute the sneak-path test cases. A realistic estimate for the complete sneak-path test suite is 315 seconds ((51 ÷ 11) × 68) in execution time.

TABLE VI. PREPARATION AND EXECUTION TIME FOR SNEAK-PATH TEST CASES

Test case generated from state machines affected by the sneak path	*Time prepare test case (sec)*	*Time execute test case (sec)*	*Mutant killed*
PreManualFS	25	7	M23
ManualFSConfirm	26	4	M18
ManualFSConfirmed	26	4	M24
AutoConfirm	25	6	M19
AutoConfirmed	25	4	M25
ExtraSlowConfirm	28	3	M16
ExtraSlowConfirm	25	4	M17
ExtraSlowConfirm	23	5	M20
ExtraSlowConfirmed	34	6	M21
ExtraSlowConfirmed	34	4	M22
ExtraSlowConfirmed	34	4	M26
Total	305	51	11

2) Effectiveness: Augmenting the conformance test suites with sneak-path testing resulted in the remaining live mutants being killed. Each and every mutant that consisted of extra, implemented behavior beyond specified behavior, was killed by the sneak-path test suite.

C. The Influence on Cost-Effectiveness when Combining Round-Trip Path and Sneak-Path

In terms of cost, we observe a large difference in test suite size; RTP contained 299 test cases whereas the sneak-path test suite contained a maximum of 68 test cases. RTP used 531 seconds on average on preparing the test suite. The sneak-path test suite was estimated to be prepared in 1,885 seconds. Furthermore, executing the RTP test suite took 489 seconds on average, and by executing the sneak-path test suite, another 315 seconds was estimated in execution time.

The effectiveness was measured by the number of mutants killed divided by the total number of mutants. By executing the RTP test suites, we saw that 58 percent (15/26) of the mutants were killed. The fault-detection ability was further improved by executing the sneak-path test suite; another 42 percent of the mutants were killed. Moreover, as the mutants that were killed by RTP and sneak-path testing were mutually exclusive, this means that RTP with sneak-path testing were fully complementary in maximizing fault detection.

V. CONCLUSIONS

In this paper, we reported on an industrial case study that investigated the cost-effectiveness of round-trip path (RTP) testing and sneak-path testing in the context of state machine-based testing.

Results show that the cost of sneak-path test suites, in terms of test-suite size, was found to be rather inexpensive as compared to the RTP coverage criteria (68 test cases for the sneak-path test suite versus a mean value of 299 test cases in the RTP test suites). By looking at the preparation time, however, the sneak-path strategy (estimate of 1,885 seconds) appeared to be more costly than the RTP strategy (531 seconds). This, however, can be explained by the construction of the test cases; the RTP test suite was generated from the complete model. The sneak-path test suite, on the other hand, was generated from 68 test models; one test model per state from the flattened state machine. Generating test cases from 68 test models as compared to one test model, required more time. In spite of differences in how the test suites were constructed, the execution time was lower for the sneak-path test suite (estimate of 315 seconds) than for the RTP test suites (489). Complementing RTP with sneak-path testing resulted in killing all remaining mutants, though at an additional cost of 1,885 seconds in preparation time and 315 seconds in execution time.

The sneak-path test suite detected the eleven remaining mutants that were not killed by any of the RTP test suites. This demonstrates that the test strategies are complementary in order to catch different types of faults. Thus, the results indicate quite strongly that *sneak-path testing is a necessary step in state-based testing (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.

Our results support the recommendation of Binder [1] and the conclusions drawn in the study of Mouchawrab *et al.* [11]: 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.

We are aware of the lack of formal statistical tests in this paper. 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. Moreover, the use of automated techniques may require changes to the original input models as to handle the tool limitations. If not carefully handled, such user interventions may pose risks related to the correctness of the modeled behavior.

ACKNOWLEDGMENT

Parts of the reported work, more specifically the development of the SUT and the field study, was funded by The Research Council of Norway through the industry project EVISOFT (EVidence based Improvement of SOFTware engineering). Lionel Briand was also supported by a PEARL grant from the Luxembourg research funding agency (FNR).

REFERENCES

[1] R.V. Binder, “Testing Object-Oriented Systems”, Addison-Wesley, 2000.

[2] T. Pender, “UML Bible”, Wiley, 2003.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “Design Patterns - Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995.

[4] S. Ali, H. Hemmati, N.E. Holt, E. Arisholm, and L.C. Briand, “Model Transformations as a Strategy to Automate Model-Based Testing: A Tool and Industrial Case Studies”, Simula Research Laboratory, Technical Report (2010-01) , 2010.

[5] T. Chow, “Testing Software Design Modeled by Finite-State Machines”, IEEE Transactions on Software Engineering, vol. 4, no. 3, pp. 178–187, 1978.

[6] N.E. Holt, E. Arisholm, and L.C. Briand, “An Eclipse Plug-in for the Flattening of Concurrency and Hierarchy in UML State Machines”, Simula Research Laboratory, Technical Report (2009-06), 2009.

[7] G. Booch, J. Rumbaugh, and I. Jacobson, The Unified Modeling Language User Guide, Addison Wesley, 1999.

[8] L.C. Briand, Y. Labiche, and Y. Wang, “Using Simulation to Empirically Investigate Test Coverage Criteria Based on Statechart”, Proceedings of the 26th International Conference on Software Engineering, 2004.

[9] Y.-S. Ma, J. Offutt, and Y.R. Kwon, “MuJava : An Automated Class Mutation System”, Software Testing, Verification and Reliability, vol. 15, no. 2, pp. 97–133, 2005.

[10] "Kermeta – Breathe Life into Your Metamodels", http://www.kermeta.org/ (May 2012).

[11] S. Mouchawrab, L.C. Briand, Y. Labiche, and M. Di Penta, “Assessing, Comparing, and Combining State Machine-Based Testing and Structural Testing: A Series of Experiments”, IEEE Transactions on Software Engineering, vol. 37, no. 2, pp. 161–187, 2011.

[12] L.C. Briand, M. Di Penta, and Y. Labiche, “Assessing and Improving State-Based Class Testing: A Series of Experiments”, IEEE Transactions on Software Engineering, vol. 30, no. 11, pp. 770–783, 2004.

[13] P. Runeson and M. Höst, “Guidelines for Conducting and Reporting Case Study Research in Software Engineering”, Empirical Software Engineering, vol. 14, no. 2, pp. 131–164, 2009.

[14] L.C Briand and Y. Labiche, “Empirical Studies of Software Testing Techniques: Challenges, Practical Strategies, and Future Research”, ACM Special Interest Group on Software Engineering Software Engineering Notes, vol. 29, no. 5, pp. 1–3, 2004.

[15] JMP, http://www.jmp.com/ (May 2012).