Assessing, Comparing, and Combining Statechart-based testing and Structural testing: An Experiment

Samar Mouchawrab, Lionel C. Briand, Yvan Labiche
Software Quality Engineering Laboratory
Carleton University, Ottawa, Canada

Abstract

An important number of studies have addressed the importance of models in software engineering, mainly in the design of robust software systems. Although models have been proven to be helpful in a number of software engineering activities, such as providing a better medium for communication among designers and customers, there is still significant resistance to model-driven development in many software organizations. The main reason is that it is perceived to be expensive and not necessarily cost-effective. This paper investigates one specific aspect of this larger problem. It addresses the impact of using statecharts for testing class clusters that exhibit a state-dependent behavior. More precisely, it reports on a controlled experiment that investigates their impact on testing fault-detection effectiveness and cost. Code-based, structural testing is compared to statechart-based testing and their combination is investigated to determine whether they are complementary. Results show that there is no significant difference between the fault detection effectiveness of the two test strategies but that they are significantly more effective when combined. This implies that a cost-effective strategy could be to specify statechart-based test cases early on, execute them when the source code becomes available, and then complete them with code-based test cases based on coverage analysis. This article also investigates the reasons for undetected faults and how the statechart-based testing of source code could be improved.

1 INTRODUCTION

There is an increasing interest in model-driven development for object-oriented systems, using for example the Unified Modeling Language (UML). In addition to be a key resource for designing object-oriented software and providing means for communicating ideas among designers and customers, models are very useful in testing object-oriented software. A number of model-based testing methodologies have been proposed based for example on use cases, class diagrams, and statecharts [8, 9, 11, 16, 34, 36, 38, 40, 41].

Model-based testing has been assessed in a number of empirical studies and showed to be useful in systematically defining test strategies and criteria, and deriving test cases and oracles [10, 12, 14, 17, 37, 38, 42]. A number of researchers conducted studies on the cost effectiveness of conventional testing strategies, i.e. white-box [21-23,

26, 46] and black-box testing strategies [44, 47] such as edge coverage and category-partition respectively, while others focused on the cost-effectiveness of model-based testing strategies such as the Round-trip path technique [10]. This related work is further detailed in Section 2.

Despite a growing number of studies [5, 9-12, 14, 16, 17, 36-38], little empirical evidence is found in literature on the importance of models in improving testing cost-effectiveness. As a result, there is little incentive for testers to adopt model-driven testing practices and it is difficult to determine how they should be integrated, if at all, with traditional testing practices. This article focuses on the effectiveness of UML statechart-based testing when compared and combined to white-box, structural testing. The main motivation for this choice is that structural coverage analysis is still the most common basic technique for testing components, but the most complex components in object-oriented software are also the ones which, according to mainstream UML development methods, should be modeled with statecharts. So assessing the cost-effectiveness of testing techniques based on statecharts and comparing it with simpler, code coverage-based techniques seems a logical investigation to undertake. The choice of UML statecharts is a practical one as UML is becoming a de facto standard. In this paper we perform both a quantitative analysis of differences in fault detection effectiveness and cost among test techniques, and a qualitative analysis to understand the reasons for these differences and the variations observed across drivers and class clusters.

At a high level, this research entails addressing the following questions:

- Are test cases identified and generated based on the statechart alone effective in detecting faults when compared to simple code-based, structural testing?
- Are the faults detected by statechart-based testing and structural testing techniques complementary?
- What are the different factors that impact the effectiveness of statechart-based testing techniques (e.g., statechart and code properties)?

Empirical studies are required to answer such questions and this can be achieved by conducting experiments on a number of object-oriented class clusters with a state-dependent behavior and their associated models. As a first step in that direction, this paper describes a controlled experiment conducted on two class clusters and which main contributions are:

- A thorough experimental evaluation of the fault detection effectiveness and cost of the state-based, Round-trip path testing technique and a comparison with simple but common baseline: simple code-based, structural testing.
- An investigation into the complementariness of statechart-based testing and structural testing to improve fault detection rates.
- An investigation of the factors that could affect the relationship between test techniques and fault detection effectiveness, including code and statechart characteristics, coverage, and the type of faults.

The paper is organized as follows: Section 2 discusses the related literature and Section 3 provides a detailed description of the conducted controlled experiment. Section

4 presents and analyses the results, while Section 4.6 summarizes the outcome of the experiment and provides learned lessons for upcoming experiments. Overall conclusions and future work are provided in Section 5.

2 RELATED WORK

Model-based testing methodologies have been proposed and advocated by researchers in a number of studies. One of the earliest works on state-based testing is the work by Chow [18] who proposed the W-method for finite state-machines. This method has been adapted to UML statecharts by Binder [8] under the name of round-trip paths strategy. In both techniques, the statechart is traversed as to construct a transition tree that includes all transitions in the statechart.

Other state-based techniques were proposed by Offutt et al. [42]. The authors introduced test techniques for generating test data from formal state-based specifications. They defined four state-based testing criteria: (1) transition coverage, (2) full predicate coverage, (3) transition-pair coverage, and (4) complete sequence. A case study was used to compare the different criteria with a random selection of test cases. Results showed an important improvement in fault detection when using the full-predicate coverage criterion. Though transition coverage yielded a small number of test cases, these test cases showed the same fault detection rate and branch coverage (of the source code control flow graph) as the random selection test strategy.

Additional testing strategies have been defined for statecharts. Hong et al. [25] propose a technique to derive extended finite state machines from statecharts. A statechart is then transformed into a flow graph modeling the control flow and data flow in the statechart thus enabling the application of conventional control and data flow analysis techniques. A modification of this method is described in [9] to address the compliance of an implementation of a system to its specification.

UML use cases were also the base for model-based system testing methodologies. Briand and Labiche in [11] proposed the TOTEM system test methodology based on use cases and their related UML artifacts including sequence diagrams, class diagrams and OCL contracts. Functional test requirements would be derived from use cases, their parameters and their sequence constraints, then transformed into test cases using the other related artifacts. An issue that encountered this methodology is the exponential increase in number of test cases when conditions, such as pre and post conditions of methods in sequence diagrams, or guard conditions include more than one predicate. This limits the automation of the methodology.

Nebut et al. defined a systematic approach for generating test-cases based on functional requirements expressed with UML use cases [37]. It is an attempt to fill-in the gap between functional specifications in form of UML use cases and concrete test cases. The authors propose a requirement-by-contract approach to add pre and post conditions to use cases. This approach is inspired by the design-by-contract approach of Meyer [35]. These contracts are used for ordering the functionalities of the system and consequently to generate correct sequence of use cases which they denote as test objective. Test scenarios are then generated from test objectives to produce executable test cases. At this

point, tester interaction may be needed to add input parameters to test cases. Note that to generate test scenarios additional information describing the scenarios corresponding to use cases are necessary (e.g. sequence diagrams). This principle of transformation has been inspired by the work of Briand and Labiche in [11]. All possible orderings of use cases are collected in one representation, the UTCS, from which a subset is selected based on one of the proposed coverage criteria as to generate test objectives. Criteria included: All edges, all vertices, all instantiated use cases, all precondition terms, and robustness criterion. The latter corresponds to exercising a use case in as many different ways as there are predicate combinations to make its precondition evaluate to false. The approach was evaluated in three case studies. Results showed that most code statements are covered by the proposed technique. The authors recommend the combination of the two criteria "all precondition terms and robustness to achieve a satisfactory trade-off between the efficiency of the obtained test set and its size [37].

A growing number of empirical studies address the cost effectiveness of testing strategies, in white-box context [21, 23, 46], black-box conventional context [47], or model-based context [2, 3, 10, 14, 37]. Many of these studies use the mutation strategy to seed faults and evaluate the fault detection effectiveness of the testing techniques. For instance, a simulation and analysis procedure [14] has been proposed and used to study the cost-effectiveness of four statechart-based coverage criteria, namely all-transitions, all-transition-pairs, full-predicate [42], and round-trip paths [8]. The results show that the cost effectiveness of testing criteria depends on the characteristics of the statechart. For complex statecharts (e.g., guard conditions), round-trip paths provide a good compromise between all-transitions and all-transition-pairs, the latter being far too expensive and the former rather ineffective.

An empirical study in the context of white-box testing strategies was performed by Frankl and Weiss [21] where the all-uses and decision (all-edges) criteria were compared to each other and to the null criterion (random test suites). This study was performed on nine very small programs whose size ranges from 33 to 60 LOCs for which the authors had access to real faults (one fault per program was used). Results showed that all-uses strategy was not always more effective than the decision and the null criterion but that when it was more effective, it usually was much more so. For decision, when more effective than the null criterion, the difference was much smaller. In other words, results varied according to the program and fault analyzed.

Briand et al. [10] focused on the cost effectiveness of the Round-trip path technique defined in [8]. The study was based on a series of controlled experiment where the Round-trip path technique was applied on a number of systems with two different levels of oracle precision. Results show that the state-based testing technique was useful in detecting faults but needed to be complemented with black-box method testing to achieve higher fault detection rates. Results of the comparison between the two oracle strategies indicate a significant difference between them in terms of fault detection and cost: precise oracles, checking the concrete state of objects after the execution of a transition, significantly increased the detection rate over state invariant oracles, though at a much higher cost.

A controlled experiment reported in [3] investigated the impact of UML documentation on software maintenance. The results showed that for complex tasks and

past a certain learning curve, the availability of UML documentation may result in significant improvements of the functional correctness of changes as well as their design quality when compared to changes produced without access to UML documentation. The results also showed that for simpler tasks, the time needed to update UML documentation was substantial compared to the potential benefits [3].

As in [10], this paper investigates statechart-based testing based on a controlled experiment. However, in addition, it assesses whether there is any gain in terms of fault detection effectiveness compared to simple test suites driven by code structural coverage. Furthermore, it explores whether the two approaches are complementary and can be combined in a cost-effective way, and what are the factors that can affect these results.

3 EXPERIMENT DESCRIPTION

In this section, we follow the template provided by Wohlin et al. in [49] to describe the experiment. First, we define the objective of the experiment and its context (Section 3.1), next we describe the plan of the experiment including the context selection criteria, the research questions and the experiment design (Section 3.2). In Section 3.3 we describe how we prepared the experiment and how we conducted it. Finally, we discuss threats to validity of the experiment in Section 3.4.

3.1 Experiment definition and context

We investigate whether statechart-based testing would somehow improve the cost-effectiveness of testing class clusters, either by itself or when combined with simple structural testing. With that goal in mind, at a high level, our dependent variables will be based on the following constructs:

- Fault detection effectiveness, overall and across different fault types to be defined in Section 3.2.5.
- Cost for both test specification and execution.

The experiment was conducted in the context of a laboratory for a fourth year engineering course on software testing. The experiment took place in the last two weeks of the course to make sure the students had gained an acceptable testing knowledge that would allow them to understand and execute the required tasks of the experiment. These students had all passed two previous courses on UML-based development and a number of courses involving Java programming.

Statechart models do not only include the statecharts themselves but also the related artifacts that are required to understand them such as class diagrams, class public interfaces (signatures, attributes), contracts and state invariants, and a textual high level description of the software functionalities and objectives. However, as subjects working with the UML artifacts are expected to use the statechart diagram to generate test cases, for the sake of brevity, we will simply refer to them as a “statechart model” in the remainder of the paper.

The experiment involved two Java class clusters; both of them have a state-driven behavior depicted in a UML statechart:

a) `OrdSet`, is a Java class (of 393 lines of code) where each instance represents a bounded, ordered set of integers. When an `OrdSet` is first created, its size gets initialized. The size of an `OrdSet` represents slots that can be used to add integers to the set. The size should be at least equal to the minimum set size and it should not exceed the maximum set size. The size of an `OrdSet` is always a multiple of the minimum set size. The user can choose a size for the `OrdSet` by providing an integer value to one of the constructors, but the actual size gets initialized based on the constraints above. The `OrdSet` class provides methods for adding a single element, removing a single element and creating the union of two ordered sets. An `OrdSet` gets resized when adding a new element if the set is full. The number of resizes allowed is set to a constant `max_accepted_resizes`. Trying to resize the set over the `max_accepted_resizes` or for a size that exceeds the maximum set size would not be allowed; in that case, an overflow in the instance of `OrdSet` is detected and no more insertion or removal of elements is allowed on the ordered set. An attempt to add or remove an element from an ordered set after an overflow is detected would raise an `OverflowException`. Figure 1 shows the statechart of class OrdSet with guard conditions in the Object Constraint Language (OCL). Some guard conditions are denoted by a letter and fully described in Appendix D to avoid cluttering the diagram.

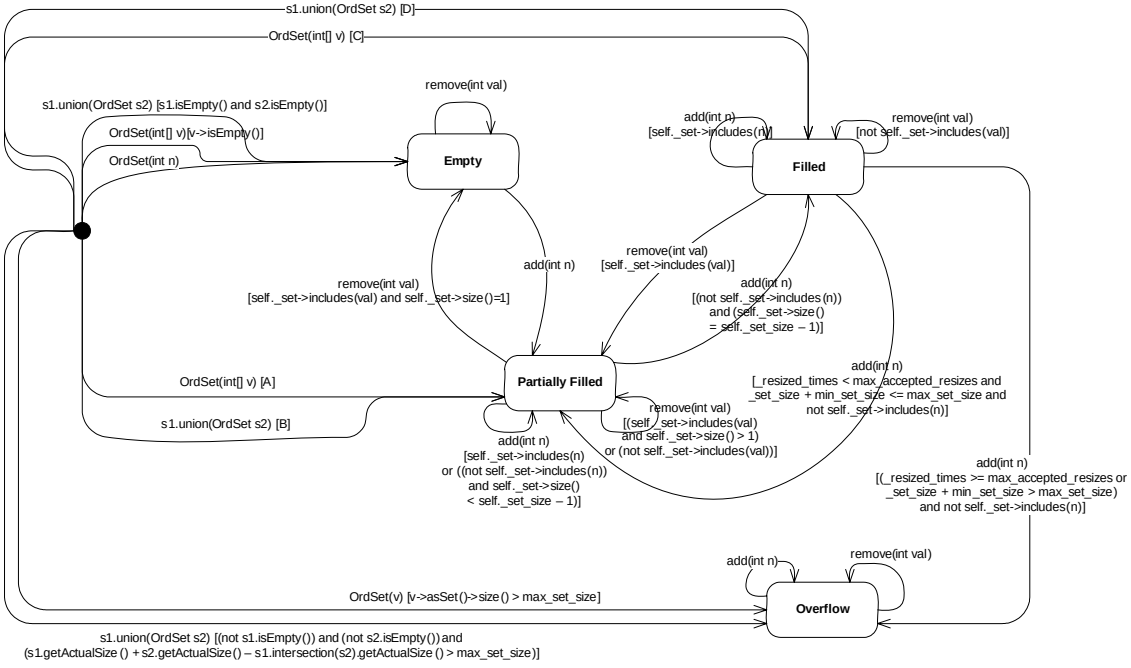


Figure 1: OrdSet state diagram

b) `Cruise Control` is a cluster of four Java classes (358 lines of code) and its class diagram and statechart are shown in Figure 2 and Figure 3, respectively. `Cruise Control` is a simplified version of a complex Cruise Control system

implemented as a 4th year engineering project. It simulates a car engine and its cruising controller, and consists of a cluster of classes:

a. *CruiseControl*: the container of the car simulator (engine simulator) and the cruise controller of that car. This is the facade of the cluster. It receives commands as strings and dispatches them to the car and the controller. The commands are: "engineOn" to start the engine, "engineOff" to stop the engine, "accelerator" to accelerate, "brake" to brake, "on" to turn the cruise control on, "off" to turn the cruise control off, and "resume" to enable the cruise control again with the earlier selected target cruise speed.
b. *CarSimulator*: simulates a car engine, runs a thread while the car is started, and simulates car speed changes based on the throttle and brake settings as well as the controlled speed by the cruising system when the latter is enabled.
c. *Controller*: simulates the cruise control of the car, it contains a *SpeedControl* thread that runs when cruising is enabled. It disables, enables or resumes cruising according to the commands received by CruiseControl.
d. *SpeedControl*: a thread that runs in the Controller to adjust car speed whenever cruising is enabled. On enabling cruising ("on"), the current car speed is recorded to be maintained for the duration of cruising time. When resuming cruising, the latest target cruise speed is used as the speed to maintain while cruising.

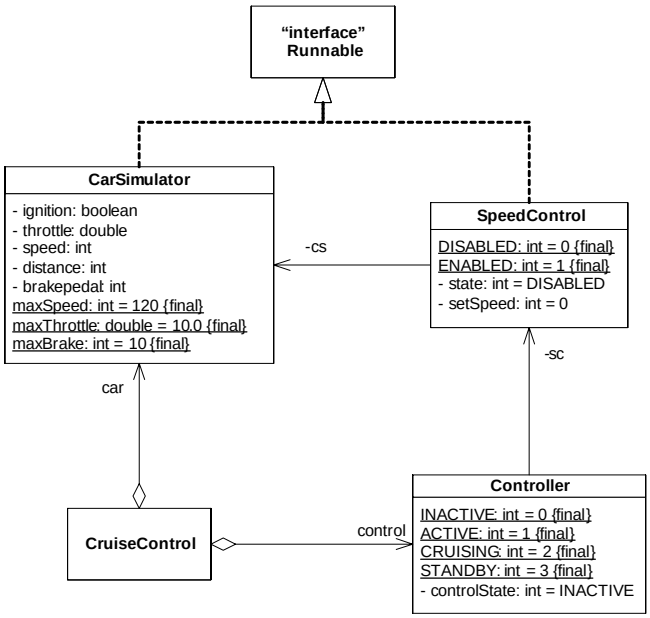


Figure 2: Cruise Control Class Diagram

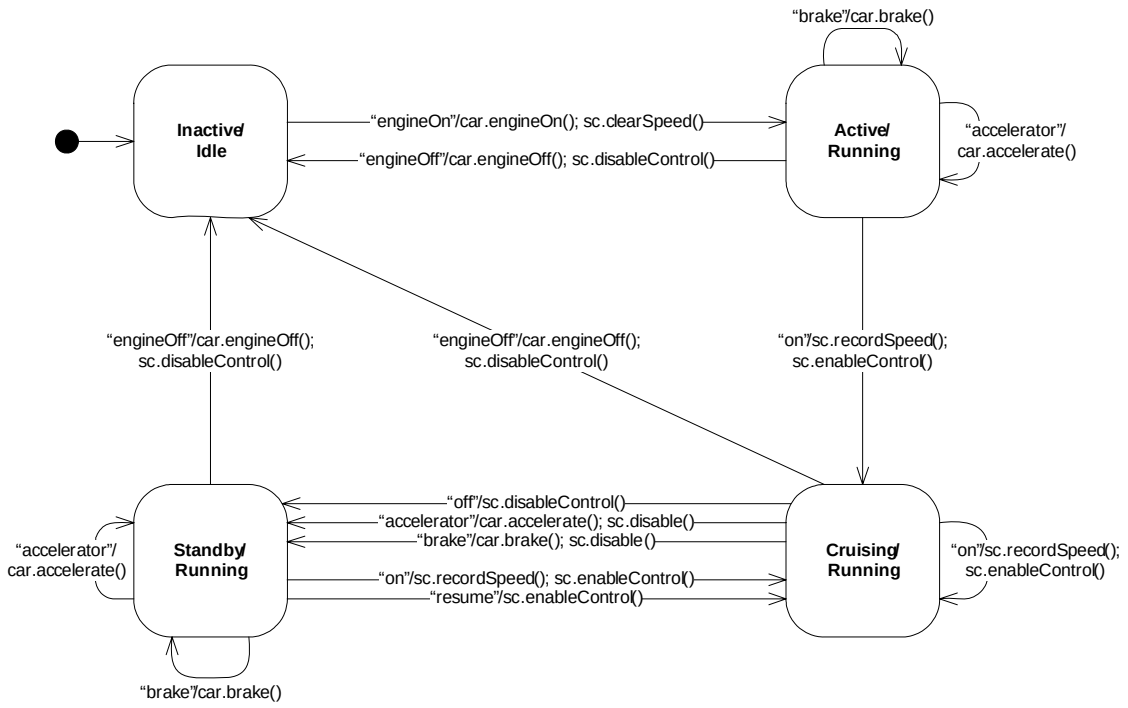


Figure 3: Cruise Control State Diagram

The two above class clusters were selected in part because of their differences. They represent two typical cases where a statechart is used to model the behavior of a complex data structure (OrdSet) and a state-dependent control class in a control system (CruiseControl). Table 1 provides size data pertaining to the two code artifacts and statecharts. Although OrdSet is composed only of one class, one can note that its statechart and control flow are more complex than those of CruiseControl; this is visible from the number of control flow edges in the OrdSet source code and the number of transitions in its state diagram. Furthermore, the guard conditions in the OrdSet statechart adds to the complexity of the class, whereas Cruise Control is event-driven only.

	Cruise Control	**OrdSet**
# classes	4	1
# operations	34	23
# attributes	14	5
# LOC	358	393
# control flow statements	33	36
# nodes	106	111
# edges	103	126
# transitions	17	22
# states	5	5
# events	7	5

Table 1: Size of source code and statecharts

3.2 Experiment planning

3.2.1 Context Selection

The subjects in this experiment were fourth-year students from a software or computer system engineering program. They were well versed in Java and UML and were attending a course on software testing that covers different white-box and black-box testing techniques with a focus on object-oriented testing. The experiment was conducted during the lab hours of that course as part of practical lab exercises. A total of 34 students participated in the experiment. The students did not know precisely what hypotheses were tested and were told that their test drivers will be marked for correctness and quality, as it would be expected from such lab exercises.

The method for the selection of subjects follows a stratified random sampling [29]; subjects were first assigned to five blocks based on their background and knowledge of object-oriented design and development techniques, then they were randomly selected from the different blocks to form four groups with a similar distribution to ensure the results would not be affected by random variations in subject experience across groups. In addition, groups were defined to be of similar sizes (Table 3) to ensure a balanced contribution of test techniques/clusters combinations to the results. However, they were practical constraints regarding the availability of certain subjects and this limited the randomization of selection. In spite of this issue, we managed to ensure that we had comparable block distributions across groups where each block is represented by a similar number of subjects in every group.

3.2.2 Research questions

In this section we provide a detailed description of the research questions to be addressed by the experiment. Table 2 lists research questions to be investigated in order to address the objectives listed in section 3.1. The fault detection effectiveness of both statechart-based and code-based test techniques is addressed in research question 1. Research questions 3 and 4 are related to studying how complementary the two techniques are. Answering research questions 2 and 5 would help us identify factors that have an interaction effect with the test technique on fault detection effectiveness while answers to research questions 7 and 8 would be used to compare test techniques in regards to their cost and cost effectiveness. In research question 6 we try to identify fault types for which one of the test techniques is a better detector, while in research question 9 we investigate the possibility of improving statechart-based testing to improve its fault detection effectiveness.

Number	Research Question
RQ1	What is the difference, in terms of fault detection effectiveness, between test cases generated from statecharts (Ts) and test cases generated only based on node and edge coverage of the source code control flow (Tc)?
RQ2	Are there interaction effects regarding fault detection effectiveness between code coverage, learning effects, subject ability and software properties (code, statechart properties) and the test technique applied?
RQ3	Are statechart-based testing and code-based testing complementary in terms of fault detection?
RQ4	When using Ts together with Tc, is there a significant improvement in terms of faults detected over using Ts and Tc alone?
RQ5	Is there an interaction effect between code characteristics of class clusters and test technique on the percentage of faults detected when combining Tc and Ts?
RQ6	Are there specific fault types that are more likely to be detected by Ts or Tc and for which the combination of both sets of test cases is particularly effective?
RQ7	How does the cost between statechart-based testing and code-based testing compare?
RQ8	How does the cost-effectiveness between statechart-based testing and code-based testing compare?
RQ9	Based on the faults not detected by Ts, what can be added to the statechart model to help generate test cases that target those types of faults?

Table 2: Research questions

3.2.3 Variable Selection

Recall the dependent variables are fault detection effectiveness and test cost. There is one independent variable of interest (treatment): The type of artifacts provided as a base to testing (i.e., statechart model or code). However, as further discussed below, a number of other variables were checked to see whether they interact with the effect of our independent variable: code coverage, learning effects, subject ability and software properties.

The treatments under investigation correspond to the following test artifacts:

a) *Code*, complemented with some textual comments to define the meaning of the most complex variables and methods. We also provide a high level textual description of the cluster objectives and functionalities.
b) *Statechart* describing the behavior of classes, plus the related public interface(s), class diagram, contracts, state invariants and a high level textual description of the software objectives and functionalities.

When statecharts are used, subjects are expected to generate test sets based on the Round-trip Path (RTP) testing technique [8], a common state-based testing strategy that can scale up to large statecharts but that is more demanding than simply covering all transitions. A statechart would be represented as a tree graph called transition tree which includes (in a piecewise manner) all the transition sequences (paths) that begin and end with the same state, as well as simple paths (i.e., sequences of transitions that contain only one iteration for any loop present in the statechart) from the initial state to the final

state. A procedure based on a breadth-first traversal of the statechart is used for deriving the transition tree. More precisely, during the traversal of the graph corresponding to the statechart, a tree node is considered terminal when the state it represents is already present anywhere in the tree or is the final state. The Round-trip Path testing technique corresponds to covering all paths from the start node to the leaf nodes in a transition tree. This tree was provided to support statechart testing in order to ensure the conformance of test suites with the RTP strategy. We thus wanted to avoid the possible effect of variations due to alternative and possibly wrong transition trees. This would have made our results more difficult to interpret and alternative transition trees are in theory supposed to be "equivalent" in the sense that they all cover (in a piecewise manner) the round trip paths.

For code-based testing, subjects were told to attempt covering all blocs (nodes, statements) and edges in the methods' control flow graphs. This is a common practice when testing classes and it is therefore a realistic baseline of comparison for the statechart-based testing technique.

For both treatments, we were aware of the fact that coverage was unlikely to be complete as time was limited and the skills of subjects were widely varying. However, we considered this was not avoidable and decided to account for it in the analysis by using coverage (statechart and code) as an interaction factor. For source code, both node and edge coverage were planned to be used in the analysis. It is often the case that controlled experiments have to choose between assessing the impact of a treatment on either the time to perform the tasks or their effectiveness, but not both [6]. We are in the latter case here.

Possible learning effects were simply measured by accounting during the data analysis for the laboratory (see next section) in which the work took place. Subject ability was measured by considering the block to which they belonged (from 1 to 5) as described in Section 3.2.1. The experiment only involved two class clusters and it is therefore not possible to analyze the impact of code and statechart characteristics on fault detection through statistical analysis. We, however, perform an in-depth, systematic qualitative analysis of why certain faults fail to be detected by test drivers.

3.2.4 Experiment design

To avoid learning or fatigue effects or the specific class clusters to have a confounded effect with our treatments, each subject group performed the experiment in two separate labs with a different class cluster under test and a different treatment. Table 3 shows the distribution of treatments among groups of subjects; the parentheses besides group numbers represent the number of subjects per group. Each treatment is executed by two different groups of subjects, in the first or the second lab (lab order). As a result, each group executed different combinations of treatment and class cluster in each lab.

Every lab lasted 3 hours. Test drivers submitted by the different subjects were executed offline on a set of mutant programs (Section 3.2.5) to measure fault detection. Test drivers were also executed on an instrumented version of the original code of the software under test to collect node and edge coverage data. The development and data collection were done on the Eclipse 3.0 platform [27]. An Eclipse plug-in, Eclipse Test and Performance Tools Platform project (TPTP) [27] was used to collect cost related

data. The specification and execution cost of a driver is assumed to be proportional to the number of methods it calls in the classes under test and we therefore use this variable as a surrogate. Though this is clearly a strong assumption, for obvious practical reasons, it has been a common one in testing studies [2, 7, 14, 26, 48]. Given that most methods in object-oriented software are small, as the number of methods called grows, this count is likely to become a more precise surrogate measure for cost.

	Group 1 (11)	**Group 2 (13)**	**Group 3 (12)**	**Group 4 (12)**
Lab 1	Cruise Control + Statechart	OrdSet + Statechart	Cruise Control + Code	OrdSet + Code
Lab 2	OrdSet + Code	Cruise Control + Code	OrdSet + Statechart	Cruise Control + Statechart

Table 3: Distribution of experiment treatments among groups

For the sake of brevity, we will refer to statechart-based testing (drivers) of the code as statechart testing (drivers). The same applies to code-based testing (drivers) which is referred to as code testing (drivers).

To address the research questions listed in Table 2, we measure the dependent variables as follows:

1. The faults detected using statecharts (Fs) and source code (Fc). The purpose here is to compare the effectiveness of statechart testing and structural testing in terms of their fault detection capability. This is involved in research question RQ1.
2. The faults detected by both test techniques ($Fs \cap Fc$). This is a measure of how redundant the two techniques are. This is involved in research question RQ3.
3. The faults detected only by statechart testing ($Fs - Fc$). We can thus evaluate the effectiveness of statechart testing to detect faults that are not detected by code drivers. This is another way to address RQ3.
4. The faults detected only by code driver ($Fc - Fs$). This helps us to identify the weaknesses and limitations of statechart testing. This is another way to address RQ3.
5. The ratio $|Fs \cap Fc| / |Fs|$. The purpose here is to evaluate the proportion of statechart test cases which are complementary to code test cases. This is another way to address RQ3.
6. The ratio $|Fs \cap Fc| / |Fc|$. The purpose here is to evaluate the proportion of code test cases which are complementary to statechart test cases. This is another way to address RQ3.
7. The faults detected when combining statechart and code test cases ($Fs \cup Fc$). The purpose here is to evaluate the effectiveness of combining techniques to overcome their limitations. This is involved in research question RQ4.

8. The ratio |Fs ∪ Fc| / |Fc|. The purpose here is to evaluate the relative improvement in fault detection resulting from combining test techniques over code test cases. This is another way to address RQ4.
9. The ratio |Fs ∪ Fc| / |Fs|. The purpose here is to evaluate the relative improvement in fault detection resulting from combining test techniques over statechart test cases. This is another way to address RQ4.
10. The variables in 3, 4, and 7 for each specific type of fault (mutation operator as discussed in the following subsection). The purpose here is to answer the above questions for each mutation operator. We only show these three variables in the analysis below as the others can be deduced from them. This is involved in research question RQ6.
11. The number of calls in test drivers to methods in classes under test (MC). The purpose here is to evaluate and compare the cost of testing strategies using a surrogate test driver size measure. This is involved in research question RQ7.
12. For each test technique cost-effectiveness is computed as the ratio of faults detected over MC. This is involved in research question RQ8.

Furthermore, a qualitative analysis of the statechart test drivers is performed to gain insights into the reasons for differences among techniques. Categories modeling possible reasons for not detecting a fault are defined and then used to classify all faults undetected by Ts. This, in turn, helps us address question RQ9.

3.2.5 Mutation operators

To compare Ts and Tc, we execute the different drivers delivered by the experiment subjects on a number of mutant programs (or mutants), that is versions of the program under test where one fault was seeded using a mutation operator [30, 31]. The mutants are generated automatically using MuJava [32]. MuJava uses two types of mutation operators, class level and traditional method level operators. The main motivations for following this procedure is to apply a systematic, automated, and independent mechanism to generate a large number of faults thus facilitating the data analysis [1]. Threats to validity related to mutation operators are discussed in section 3.4.

One issue to be addressed is the detection of equivalent mutants, i.e. mutants that have the same behavior as the original program and therefore cannot be killed by test cases. Manually identifying equivalent mutants is the most common practice but is time consuming and error-prone. A number of studies addressed this issue and proposed optimization techniques to automate the detection of equivalent mutants [39, 43]. However, these methods have shown to detect on average only half of the equivalent mutants, and the detection ratio depends heavily on the program characteristics [43]. Instead, some authors proposed, as a heuristic, to consider live mutants not killed by any test case in the overall test pool as equivalent mutants [4, 15, 19]. This approximation is thought to be good enough especially when dealing with large number of mutants. But in our case we know the testing performed by our experiment subjects is incomplete and unlikely to kill all non-equivalent mutants. Therefore, we do not attempt to discard equivalent mutants but present our results based on all mutants and then perform a

manual, qualitative analysis of all undetected faults (Section 4.4) to assess the potential impact of equivalent mutants on the fault detection effectiveness results.

Table 4 includes the list of all mutation operators used in this experiment along with a brief description.

Mutation Operator	Level	Description
AORB (Arithmetic Operator Replacement – Binary)	method	Replaces basic binary arithmetic operators with other binary arithmetic operators.
AORS (Arithmetic Operator Replacement – Short-cut)	method	Replaces short-cut arithmetic operators (++, --) with other unary arithmetic operators.
AOIU (Arithmetic Operator Insertion – Unary)	method	Inserts basic unary arithmetic operators.
AOIS (Arithmetic Operator Insertion – Short-cut)	method	Inserts short-cut arithmetic operators.
AODU (Arithmetic Operator Deletion – Unary)	method	Deletes basic unary arithmetic operators.
ASRS (Assignment Operator Replacement – Short-Cut)	method	Replaces short-cut assignment operators (+=, -=, *=, /=, %=) with other short-cut operators of the same kind.
ROR (Relational Operator Replacement)	method	Replaces relational operators with other relational operators.
COR (Conditional Operator Replacement)	method	Replaces binary conditional operators with other binary conditional operators.
COD (Conditional Operator Deletion)	method	Deletes unary conditional operators.
COI (Conditional Operator Insertion)	method	Inserts unary conditional operators.
LOI (Logical Operator Insertion)	method	Inserts unary logical operator.
IOD (Inheritance – Overriding method Deletion)	class	Deletes an entire declaration of an over-riding method in a subclass so that references to the method use the parent's version.
JDC (Java-supported Default constructor Creation)	class	Forces Java to create a default constructor by deleting the implemented default constructor.
JID (Java – member variable Initialization Deletion)	class	Removes the initialization of member variables in the variable declaration so that member variables are initialized to the appropriate default values of Java.
JSI (Java – Static modifier Insertion)	class	Adds the static modifier to change instance variables to class variables.
JSD (Java – Static modifier Deletion)	class	Removes the static modifier to change class variables to instance variables.
EAM (Encapsulation – Accessor Method change)	class	Changes an accessor method name for other compatible accessor method names, where *compatible* means that the signatures are the same.

Table 4: Mutation operators

Figure 4 shows the distribution of the created mutants among the different mutation operators for the two clusters. This distribution looks different for the two class clusters

under test due to differing code characteristics. For example, 11 mutants have been created with the AORS mutation operator (see Table 4) for OrdSet, and none for Cruise Control which has no shortcut arithmetic operators, i.e. ++ and --; 12 mutants have been created with the ASRS mutation operator (see Table 4) for Cruise Control, and none for OrdSet which has no shortcut assignment operator, i.e. +=, -=, /=, *= and %=.

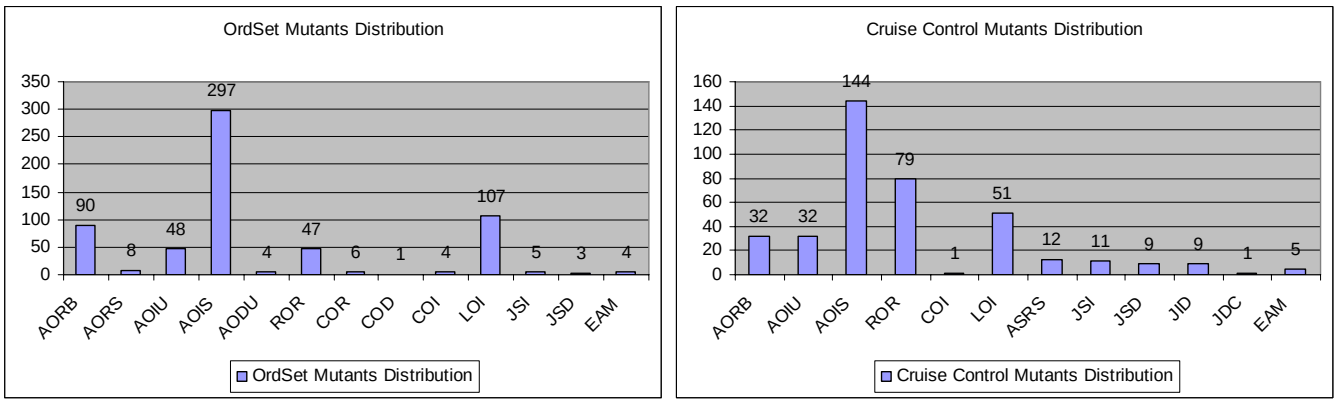


Figure 4: Mutant distributions

3.2.6 Overview of Statistical Analysis

A variety of statistical techniques were applied to address research questions and we provide here a short overview, leaving the details for the section presenting the experimental results.

Univariate analysis was performed to compare the isolated effect of an independent variable on a dependent variable. For example, two-sample *t*-tests were performed to compare test techniques in terms of fault-detection effectiveness and cost, and determine whether differences in means could be due to chance. The level of significance is set to $\alpha = 0.05$ for all tests, though we also report p-values. To avoid potential threats due to the violation of the *t*-test assumptions, equivalent non-parametric tests (Wilcoxon rank sum tests [20]) were also performed and in the rare cases where differences of results can be observed, this is clearly stated.

To help visualize results, we use both means diamonds and box plots. A mean diamond indicates the sample's mean and 95% confidence interval and whether this is significantly different from other samples. Box Plots show selected quantiles of distributions and extreme values.

Regarding multivariate analysis, depending on the covariates involved, we either perform a two-way analysis of variance (ANOVA) or a bivariate least-squares regression [20] to study the simultaneous effect of the test technique on fault detection and its interactions with other factors (e.g., coverage). This is important as the effect of test techniques can vary widely based on factors related to class cluster and statechart characteristics, subject ability, and so on.

3.3 Experiment operation

3.3.1 Preparation

The students were first introduced to the class clusters under test during the experiment to make sure they solely relied on the documentation presented to them. To prepare the students for the different tasks required for the experiment, they were also given a refresher on the basics of testing (test cases, test sets, testing criteria, drivers …), structural and functional testing, and class testing. Students applied the concepts and techniques they were taught in assignments on laboratory exercises prior to the start of the experiment's tasks.

To calculate node and edge coverage, the classes under test were instrumented using the Observer pattern [24] and by building the control flow graphs of their methods. The instrumentation code includes the definition of control flow nodes and edges, an Observer class that is informed of visited nodes and edges, and a Recorder class that generates coverage report.

Each of the two labs lasted 3 hours, during which students were provided documentation and executable code to run their drivers on, and asked to write driver code following precise instructions. The following documents were provided to all students in all groups:

1. Printed list of instructions to guide students through the different tasks to complete.
2. High-level description of the cluster.
3. Eclipse tutorial.
4. Driver template (differs slightly depending on the testing strategy).

For groups working with statecharts, the following documents were also provided:

1. Class public interfaces
2. Model documentation including class and statechart diagrams, operation' contracts and state invariants in OCL, and a transition tree.
3. An executable jar file of the class cluster.

For the groups working on code testing an instrumented version of the code in a form of an executable jar file was provided along with the original non-instrumented source code of the class clusters.

3.3.2 Execution

During each lab, students were first asked to read the documentation of the class cluster to understand its functionalities; then they were asked to identify test cases based either on the provided transition tree (covering round-trip paths) or based on all-nodes and all-edges structural coverage criteria, depending on the group they belonged to. In the latter case, students were asked to write method sequences capturing realistic scenarios in their test cases, and they were advised to use Equivalence class testing or boundary analysis [28, 50] to help the identification of method parameter values. For code testing,

students were instructed to run their drivers on the instrumented version of the code to compute node and edge coverage; the generated report identifies the non-covered nodes and edges, which can guide students to identify new test cases to be added to their drivers to improve structural coverage.

When applying statechart testing, students were instructed to use the common practice of state invariant assertions as oracles for their test cases; for code testing students were advised to write oracles checking expected output/attribute values against actual ones; it was recommended to add an oracle after each method execution in the method sequences to verify the validity of the outputs and changes to attribute values.

After all test drivers were submitted to us by participants, we executed them on the original code of the two class clusters to inspect their correctness and to eliminate any inadequate drivers which could not be used for experimental purposes (such as drivers with no oracles).

Perl scripts were created to automatically execute drivers on mutants and on code instrumented versions, in order to collect data required to compute mutation scores, node and edge coverage, and distributions of undetected fault types.

For cost-related data, the profiling tool of Eclipse plug-in TPTP [27] was used to count the number of method calls. The notion of cost in the context of testing can be related to many factors such as test size, test case identification complexity, computer time usage and time to market. In many studies, the size of a test set (i.e., the number of test cases) has been adopted as a surrogate measure for cost [14, 26, 46], assuming that cost is overall proportional to test set size. In these studies, one test case often corresponds to one execution of a function/program (e.g., [26]). This corresponds in our study to one execution of a cluster method. We therefore use the number of method calls in a test set as a surrogate for cost.

3.4 Threats to validity

A brief summary of threats to validity [49] in our experiment is provided below.

3.4.1 Conclusion validity

Threats to conclusion validity are concerned with issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome of the experiment [49]. The threats to conclusion validity in our experiment could be related to: (1) low statistical power and (2) reliability of treatment implementation.

Regarding (1), we were limited by the number of students enrolled in the testing course within which we conducted the experiment. To limit the impact of this threat on our conclusions, we designed the experiment in such a way that each group would work on a different treatment for two successive labs and thus doubled the number of observations.

As for the reliability of the treatment, for code testing, there was significant variation in structural coverage for code testing. This is not the case for statechart testing as subjects were asked to implement test cases based on a given transition tree. The round-trip path technique is systematic: using one transition tree should result in a similar

outcome and this is especially true for the Cruise Control where no arguments are required to implement test sequences corresponding to paths in the transition tree. Though this problem does impact the fault-detection effectiveness of code drivers, we account for it when performing a bivariate analysis using both variables testing technique and coverage.

3.4.2 Internal Validity

An internal validity threat exists when the outcome of the experiment may not necessarily be caused by the treatment applied but caused by another factor not controlled in the experiment. One example of internal validity threats is the learning and fatigue effects that can occur when the experiment is run more than once with the same subject groups. This threat is addressed in our experiment by using different treatments and different class clusters in each of the labs conducted for each group.

To tackle the selection threat that is related to the variation in human performance, we identified a number of blocks to which correspond the students. These blocks are based on students marks achieved in earlier courses on software engineering and design. Students were selected from the different blocks to have a stratified random sampling over the different groups.

Another internal validity threat, the diffusion or imitation of treatments, was also limited by monitoring the labs and preventing the access to the experiment documentation outside the lab hours and by other groups' members. Note that the experiment documentation is accessed through the course website only during lab hours with an address only known during the lab by members of the group working in that specific lab.

To the difference with subjects working with the code treatment, those working with the statechart treatment were not instructed to use the Equivalence classes or boundary analysis to identify test cases parameters values. This threat is eliminated with the fact that for statechart testing in this experiment, no boundary analysis was needed: the Cruise Control's statechart has no parameters, and boundary analysis of the OrdSet class was accounted for in the guard conditions in statechart as to eliminate any bias of using an additional testing technique with the statechart technique to be implemented.

3.4.3 Construct validity

The construct validity in our case is related to the fact that we used mutation analysis to measure the fault-detection effectiveness of testing strategies. The types of faults seeded may not be representative of "real" faults. To limit the likelihood for this threat to manifest itself, we used two class clusters with very different code characteristics (Section 3.1). Also, the results of the study reported in [1] show that faults seeded using mutation operators can be representative of real-faults when measuring fault detection effectiveness.

3.4.4 External Validity

External validity relates to the external aspects that interact with the treatments and limit the generalization of the results. The selection of fourth-year engineering students as subjects could be a threat to external validity as they could not be representative of "real" software developers. However, first we do not believe there is such a thing as a "real"

software developer population. It is well-known that productivity can vary a level of magnitude between the best and worst developers. Second, these students were on average good Java developers as this is the main language used throughout their four years of study and they are better acquainted with UML than most average practitioners since they undertook two full term courses on the subject. So, overall, for the specific tasks at hand, these students are probably comparable to at least average practitioners.

The choice of the clusters to test in this experiment may be considered an external validity threat. However, while simple and small, the class clusters used in our experiment still can be considered representative of two common types of class clusters. The Cruise control is representative of real-time, reactive classes with a state-dependent behavior where the different class attributes are evaluated based on elapsed time between events and the current cluster state. The OrdSet class, on the other hand, is modeled by a large statechart with complex guard conditions. It is representative of classes encapsulating complex data structures with large transition trees (30 Round-trip paths).

4 EXPERIMENT RESULTS

In the first section, we report on the drivers' mutation scores and code coverage for the two testing techniques. Then, we perform an analysis to determine the impact of the test techniques and other factors such as code coverage and cluster characteristics on fault detection effectiveness (Section 4.1). This analysis aims at answering research questions 1 and 2 (Table 2). Next, in Section 4.2, we answer research questions 3, 4, 5 and 6 by investigating the complementariness of test techniques, their combination, and its impact on fault detection effectiveness. We further investigate the impact of combining test techniques on the fault detection effectiveness per mutation operator and we identify fault types for which statechart test cases are better detectors (research question 6). Cost analysis and cost-effectiveness analysis in Section 4.3 address research questions 7 and 8. In Section 4.4, a qualitative analysis investigates live mutants in order to determine the reasons for not detecting seeded faults and to understand the limitations of the statechart test technique. In the subsequent section (4.5) and based on the results of the qualitative analysis, improvements to the statechart test technique are proposed to increase fault detection with statechart drivers (research question 9).

4.1 Impact of test techniques on fault detection effectiveness

We discuss in this section the impact of the independent variable "test technique" (code vs. statechart) on the dependent variable "fault-detection effectiveness" which is measured as a mutation score (Section 4.1.1). Next, we investigate the possible interactions between the test technique and a number of factors and their impact on fault detection effectiveness (Section 4.1.2). These factors include: code coverage, lab order, subject ability and cluster characteristics.

4.1.1 Univariate analysis

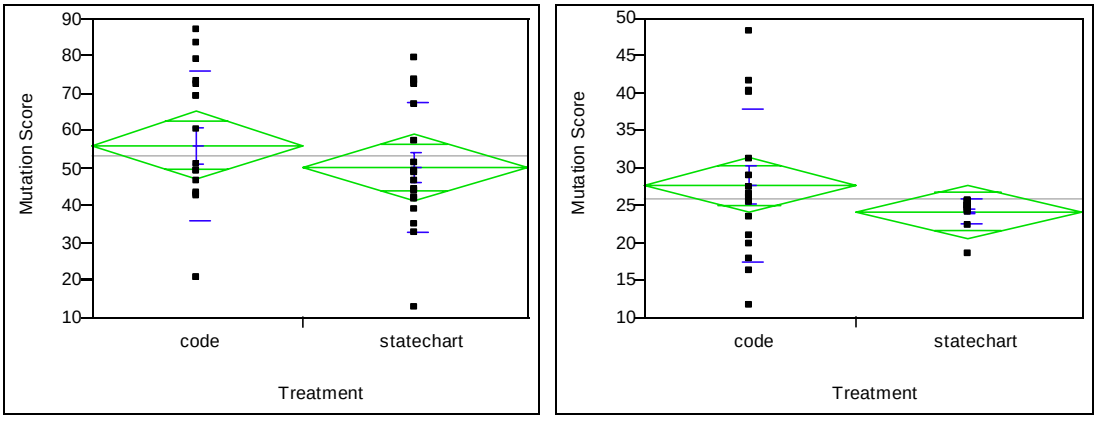
Table 5 provides descriptive statistics of the mutation scores in each system and for both treatments. A graphical representation of mutation scores distribution is provided in Figure 5 where mean diamonds of the mutation scores obtained for test drivers are depicted. Results show that the maximum and mean mutation scores for both systems

were higher for code drivers than for statechart drivers. For the Cruise Control, this is mainly related to the real-time properties of the code. The statechart of Cruise Control (Figure 3) does not model its real-time behavior, thus subjects working with the statechart had no access to a description of how values of class attributes such as "car speed", "throttle" and "total distance" are calculated and updated over time. These values depend on many factors such as time and air resistance and they are constantly changing when the car is running in the "active" state. An activity diagram as the one provided in Figure 6 is more suitable to describe the real-time behavior of Cruise Control. For OrdSet, the difference in mutation scores of code drivers and statechart drivers can be explained by the fact that subjects working with code were provided an instrumented version of the code allowing them to identify uncovered nodes and edges and to write test cases that address them.

Cluster	Treatment	Min	Max	Mean	StdDev	Range
OrdSet	Statechart	12.66	79.17	50.27	17.20	66.51
	Code	20.35	86.7	56.15	19.98	66.35
Cruise Control	Statechart	18.39	27.46	24.47	1.65	9.07
	Code	11.4	48.19	27.69	10.24	36.79

Table 5: Mutation scores descriptive statistics

Results also show that Cruise Control drivers for both code and statechart testing had low mutation scores compared to OrdSet drivers' mutation scores. This is again likely due to the real-time behavior of Cruise Control. As mentioned, subjects working with Cruise Control's statechart had no access to documentation on the real-time behavior of the cluster and did not manage to exercise parts of the code. Although subjects working with the Cruise Control code noticed the importance of varying time in their drivers, it was hard for them to understand the real-time algorithm that manages the class attributes solely based on code. A thorough understanding would have required complex reverse engineering or access to documentation such as the activity diagram in Figure 6 that models the real-time algorithm control-flow managing a running car.



a) OrdSet b) Cruise Control

Figure 5: Mutation scores' distribution

Other noteworthy results from Table 5 are the standard deviations in mutation scores for the different drivers of a given cluster under test and test technique. For Cruise

Control, the standard deviation for statechart drivers is small (less than 2%). This can be easily explained: (1) the testing criterion is well defined and leaves little degree of freedom in its application (one must cover all Round trip paths), (2) the transition tree used is the same for all subjects: a decision we made to ensure the conformance of test suites with a correct transition tree [13] and (3) transitions in the statechart have no guard conditions and require no parameter setting. Therefore, by following the RTP technique, similar results should be obtained by all subjects causing a small standard deviation. The differences in mutation scores are related to wrong or incomplete implementation of state invariants in oracles, or are due to the incomplete coverage of the transition tree.

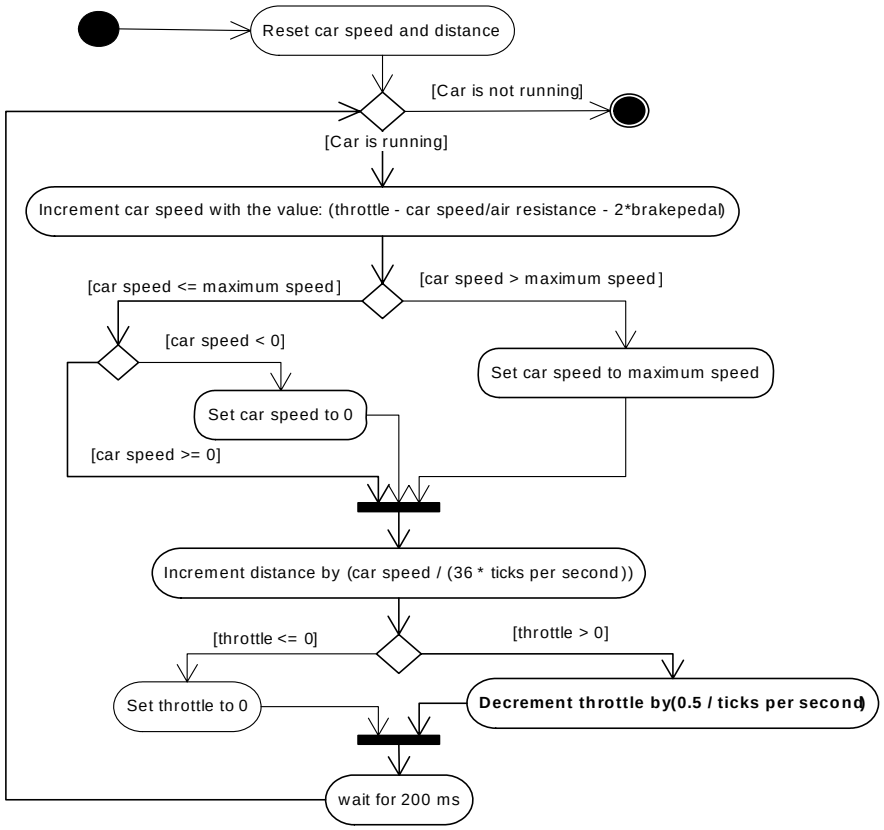


Figure 6: Cruise Control - Running car activity diagram

As opposed to Cruise Control, the mutation scores' standard deviation for the OrdSet statechart drivers is fairly large (17% as opposed to 1.5% for Cruise Control). This can be easily explained: (a) only few subjects were able to cover all RTPs (35% RTP coverage on average) as test cases in their drivers covered various numbers of RTPs (see Table 6 for code and RTP coverage descriptive statistics), (b) the statechart has complex guard conditions and requires parameter settings which ten introduce variation in test cases, (c) some faults can be detected only with very specific parameter values or set content, and (d) wrong or incomplete implementation of state invariants in oracles. We assume that if testers had unlimited time to complete the implementation of all round trip paths as described in the test technique, the standard deviation would decrease and only depend on points (c) and (d) listed above.

Table 6 provides descriptive statistics on node, edge, and RTP coverage for both class clusters. Results show that statechart drivers have lower node and edge coverage than code drivers. The difference, which is relatively small (e.g., 7% in edge coverage for Cruise Control), can be explained by the fact that subjects working with code used node and edge coverage analysis to refine their test drivers and achieve better coverage. As for RTP coverage, OrdSet had much lower RTP coverage than Cruise Control (35% as opposed to 97%). This is mainly due to the complexity of the OrdSet statechart: 30 RTPs as opposed to 12 for Cruise Control and, in addition, complex guard conditions.

	Cruise Control					**OrdSet**				
	Statechart			**Code**		**Statechart**			**Code**	
Coverage	**RTP**	**Node**	**Edge**	**Node**	**Edge**	**RTP**	**Node**	**Edge**	**Node**	**Edge**
Median	100	85.85	69.9	86.79	75.73	45	71.17	61.90	81.08	73.02
Mean	96.59	84.49	68.93	87.21	75.61	34.81	76.93	67.50	81.08	73.53
95%	100	94.34	79.61	97.17	94.17	100	94.59	90.48	99.10	95.40
90%	100	87.54	74.37	96.51	91.45	100	92.70	86.59	99.10	94.76
75%	100	85.8	71.1	93.2	83.5	85	84.91	76.19	97.30	91.27
25%	100	82.78	68.2	81.13	68.2	35	68.47	57.54	73.87	65.87
10%	82.5	78.3	59.1	77.1	61.5	19	66.13	51.90	54.78	44.92
5%	53.75	78.3	58.3	76.42	55.34	10	64.73	50.43	46.31	36.03
Min	50	78.3	58.25	76.42	55.34	10	63.96	48.41	40.54	30.95
Max	100	94.34	79.61	97.17	94.17	100	94.59	90.48	99.10	96.03

Table 6: Code and statechart coverage descriptive statistics

For both test techniques, mutation scores were definitely much higher for OrdSet than for Cruise Control. However, node and edge coverage for both class clusters were comparable for both test techniques (Table 6). This can be explained by the fact that for Cruise Control, most changes to class attributes such as “speed”, “distance” and “throttle” are computed in two methods (i.e., limited number of edges and nodes), which are the “run” methods of threads representing the car and its speed controller. Although these two methods are only of few lines of code, they include a significant number of computation and assignment statements for which the number of generated mutants is high. Many of these statements were not covered by statechart drivers because of their limited time of driver execution, due to a lack of understanding of real-time properties of the code.

As discussed above, descriptive statistics show a difference in terms of percentage of faults detected (mutation scores) between the two test techniques. We performed a two-sample *t*-test [20] for each class cluster to assess the statistical significance of this difference. For research question 1 (Table 2) we tested the following null hypothesis: “There is no significant difference between the number of faults detected by statechart test cases (Ts) and code test cases (Tc)”. The results are reported in Table 7. For both clusters a *t*-test yielded a *p* value greater than $\alpha = 0.05$ and therefore the null hypothesis cannot be rejected. No statistically significant difference in terms of mutation scores of

statechart drivers and code drivers can be observed and we cannot therefore claim that one type of drivers is more effective at detecting faults than the other.

Cluster	DF	Mutation score mean		t-value	Pr > |t|
		Code	Statechart		
OrdSet	31.6	56.15	50.27	0.93	0.359
Cruise Control	15.8	27.69	24.47	1.35	0.197

Table 7: *t*-test results for the mutation score comparison

4.1.2 Interaction effects

We study in this section the interaction effect with the test technique on mutation score of a number of factors: code coverage, cluster, lab order, and subject ability..

a. Code coverage impact

Figure 7 shows a comparison of mutation scores for code and statechart drivers with similar node and edge coverage. Drivers are grouped when in the same 5% coverage interval. For example, in Figure 7 we denote the percentage interval]75, 80] as “80” and proceed the same way for subsequent intervals. Note that only a subset of intervals contain data points as there is no driver for certain levels of coverage. For Cruise Control, similar coverage values yielded overall similar mutation scores when comparing code and statechart drivers. For OrdSet, code drivers with low coverage rates show better fault detection rates than statechart drivers. This is due to the fact that the few statechart drivers having low code coverage have covered only a small subset of RTPs in their drivers, or they did not implement the test strategy as instructed, i.e. did not implement state invariant assertions in oracles. This suggests that when only a small subset of RTPs is implemented in the driver, it somehow contains RTPs that are less likely to detect faults than the "equivalent" code driver. Further investigation showed the first RTPs implemented were the simplest ones, and therefore the least likely to detect faults. But, for high coverage rates similar coverage yielded similar mutation scores.

The similarity in node and edge curves suggests a linear dependency between the two variables. This is confirmed in Figure 8 where the linear fit line of the two variables is shown. The proportion of the variation that can be attributed to terms in the model rather than to random error ($R^2 = 0.9$). Therefore, 90% of the variability in edge coverage can be attributed to node coverage. Based on this result, we limit any subsequent analysis to node coverage.

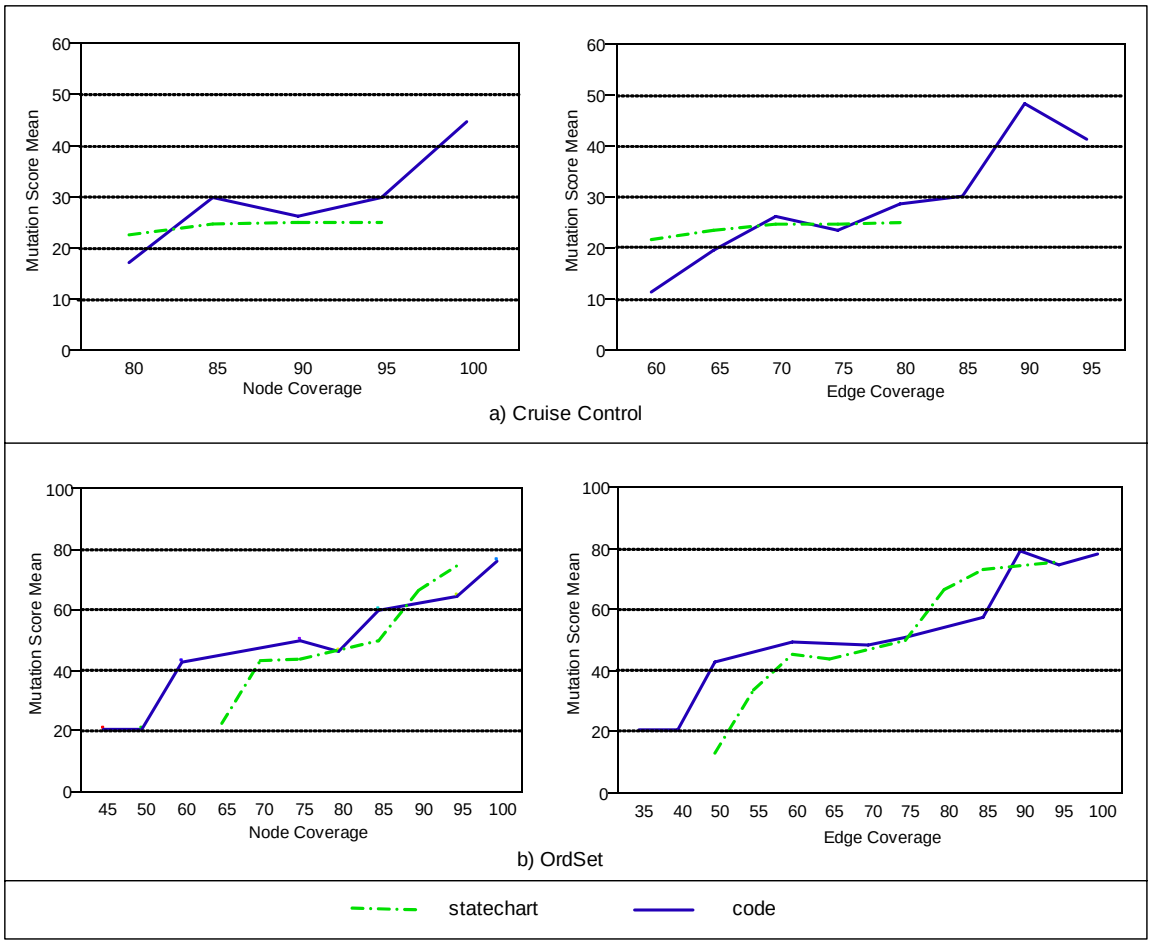


Figure 7: Mean mutation scores of code and statechart drivers as a function of node and edge coverage

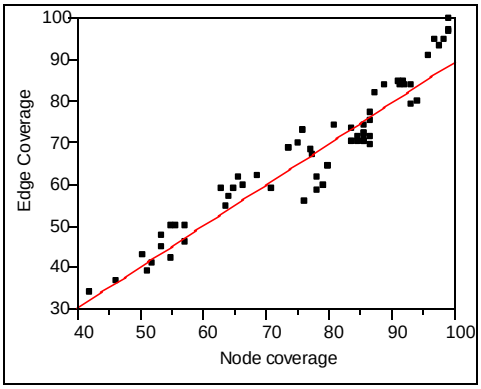


Figure 8: Node - Edge coverage linear dependency

A multiple regression analysis was performed to assess the impact of node coverage and its interaction effect with test technique on mutation scores. The results presented in Table 8 show a significant main effect of node coverage (or in general code coverage) on mutation scores ($p < .0001$), but also a significant interaction effect of node coverage and test technique. Figure 9 shows interaction plots for Node coverage and Test

technique for both clusters. For Cruise Control, when code coverage increases, code drivers tend to have higher mutation scores than statechart drivers. This can be explained by the fact subjects working with statecharts had no access to documentation on the real-time behavior, and therefore did not explicitly target real-time faults and certain parts of the code. However, for low code coverage, statechart drivers show higher mutation scores. This is likely due to the use of state invariants assertions in oracles.

As for OrdSet, the trend is reversed and statechart drivers show higher mutation scores for high code coverage rates. With precise and detailed guard conditions, the OrdSet statechart provides enough information for its users to cover the code to a large extent. This is different from the Cruise Control statechart where, even when fully covering the transition tree, one is unlikely to cover large parts of the code without precisely understanding the real-time properties.

Cluster	RSquare	Factor	Sum of Squares	Parameter estimate	F Ratio	Prob > F
Cruise Control	0.61	Node Coverage	930.49	1.213	40.31	**<.0001**
		Test technique	20.92	-1.648	0.90	0.3489
		Node Cov * Test technique	140.65	-0.927	6.09	**0.0197**
OrdSet	0.78	Node Coverage	4985.58	0.901	59.74	**<.0001**
		Test technique	1395.79	15.684	16.72	0.0003
		Node Cov * Test technique	722.83	0.821	8.66	**0.0061**

Table 8: ANOVA - Impact of node coverage and its interaction with test technique on mutation scores

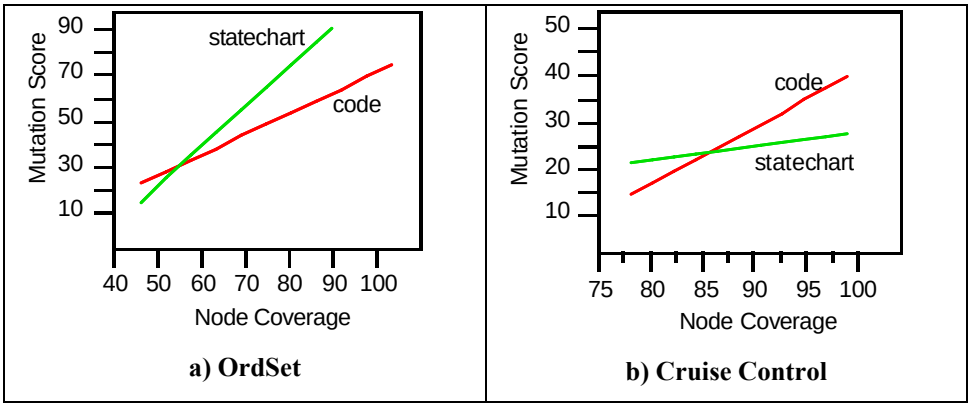


Figure 9 : (Node Coverage x Test technique) interaction plots

b. Lab Order impact

In this subsection we study the main and interaction effects of lab order (learning effects) on mutation scores to account for learning effects. It is assumed that subjects would tend to perform better on the second lab than in the first lab, regardless of other factors. An analysis of variance (ANOVA) was performed and the results are presented in Table 9. Results showed no significant impact of lab order, either direct or through interactions, on mutation scores. A plausible reason is that subjects were well trained for the tasks from the start and learning effects were therefore limited.

Cluster	RSquare	Factor	Sum of Squares	Parameter estimate	F Ratio	Prob > F
Cruise Control	0.08	Lab Order	33.95	-1.456	0.60	0.4444
		Test technique	101.67	-3.565	1.80	0.1903
		Lab Order * Test technique	27.26	1.846	0.48	0.4927
OrdSet	0.099	Lab Order	510.38	-5.488	1.54	0.2239
		Test technique	509.90	-7.758	1.54	0.2241
		Lab Order * Test technique	79.18	3.057	0.23	0.6283

Table 9 : ANOVA - Impact of lab order and its interaction with Test technique on mutation scores

c. Subject ability impact

Subject ability was measured based on grades in software design courses. When subject's ability increases, mutation score would a priori be expected to increase as well. But Figure 10—which shows means, min and max values as well as 95% and 5% percentiles for mutations cores—provides a partially different picture. Although for Blocks 1 to 4 (decreasing order of ability) the mutation score decreases as expected, subjects in block 5 (supposedly the least skilled) developed drivers with high mutation scores. On average, their drivers had mutation scores close to those written by the most skilled subjects. To explain what might have happened, recall that blocks were based on the ability of students in software design (i.e., final marks in software design courses). The results in block 5 suggest that the method used for evaluating subject ability may not be optimal for this experiment as skills in programming and testing may not be entirely correlated to those of software design. The trend may also be due to the fact that our groups are small and may therefore be strongly and randomly affected by outliers. Checking more closely at the distribution of scores in group 5 we indeed see such outliers, though we have no explanations for them, except perhaps that our ability measurement may not be fully adequate.

Though we used five blocks for the purpose of random subject assignment (Section 3.2.4), in order to have large enough samples at each ability level (thus alleviating the outlier problem) and to ensure a balanced design to enable the use of ANOVA, we only use two ability levels to analyze the effects of ability on mutation score. We assign all subjects to either a HighAbility or LowAbility group, depending on whether the grade on which the blocks are defined was below or above the median grade[1].

The results of a two-way ANOVA with subject ability (2 levels) and its interaction with test technique on mutation scores are reported in Table 10. Results show that subject ability has significant main effect and a marginally significant interaction effect (slightly above 0.05) with the test technique for Cruise Control. Furthermore, as opposed to univariate results in Table 7, the impact of Test technique is also significant and similar in the variance it explains to that of Ability: code-based testing yields significantly higher

[1] It is equivalent to merging blocks 1&2 and 3 to 5, respectively.

scores. This is not the case for OrdSet where subject ability or test technique have no effect on mutation scores. One plausible explanation is that despite the complexity of its statechart, the functionality of OrdSet is rather intuitive for engineering students and then the ability to understand the statechart and code were not as crucial as for CruiseControl. For Test technique, the need to understand code properties is not as crucial for OrdSet as the statechart is an accurate description of its behavior.

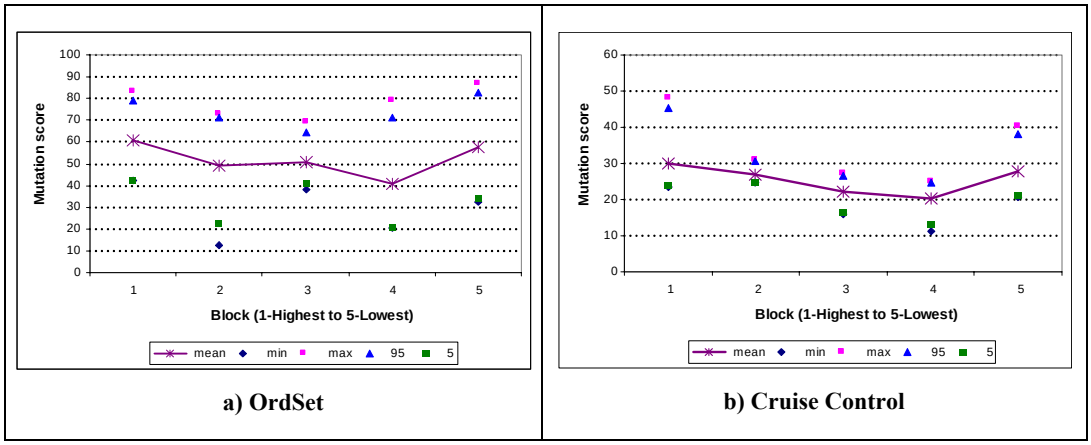


Figure 10: Relationship between subject skill and mutation score

Cluster	Factor	Sum of Squares	Parameter Estimate	F Ratio	Prob > F
Cruise Control	Subject ability	281.08	-5.93	6.76	**0.0147**
	Test technique	268.71	4.10	6.46	**0.01687**
	Subject ability * Test technique	171.59	-4.63	4.13	0.05177
OrdSet	Subject ability	358.02	-6.50	1.05	0.31257
	Test technique	292.83	4.16	0.86	0.3603
	Subject ability * Test technique	1.06	-0.35	0.003	0.9558

Table 10 : ANOVA - Impact of subject ability and its interaction with test technique on mutation scores

Recall we have looked at the effect of Ability by grouping Blocks 1 & 2. If we now look at Block 1 (20% best subjects) in isolation (HighAbility = Block 1) and run ANOVA gain, we obtain the results shown in Table 11. Because the number of observations is not balanced anymore across Ability/Test Technique categories, the order in which variables are introduced in the ANOVA model matters. We have to perform sequential testing and select an order that makes sense: we estimate the contribution of each variable in order, where we compute the sum of squares explained by Ability and then in turn allocate the remaining sum of squares to Test technique. Once we have accounted for main effects we then estimate the impact of interaction effects between Ability and Test technique by computing the remaining sum of squares it accounts for. For OrdSet, the results do not change. But for CruiseControl, Ability still has a significant main effect but it has a much stronger interaction effect.

Figure 11 shows an interaction plot of subject ability and test technique for Cruise Control when HighAbility = Block 1. It clearly shows that when following a code coverage test strategy, subject ability has an important effect on mutation scores:

mutation scores increase when subject ability increases. However, for statechart drivers, mutation scores were hardly affected by subject ability. This suggests that having a clear test model like statecharts and a precise test strategy like RTP alleviates the impact of the tester's ability on test results. Test effectiveness is more predictable with statechart but the higher skilled subjects, when having access to code, would identify test cases that cannot be easily identified with statechart RTP testing and therefore perform better. An example of such test cases would be inserting waiting times in test drivers to allow for boundary testing (e.g., maximum speed).

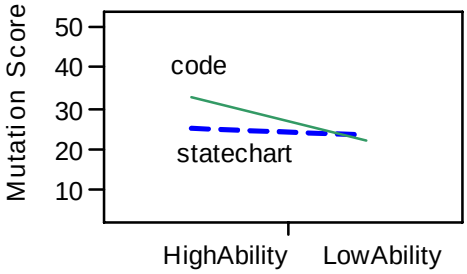


Figure 11: (Subject Ability x Test technique) Interaction plot - Cruise Control

Cluster	Factor	Seq SS	F Ratio	Prob > F
Cruise Control	Subject ability	256.23	7.018	**0.0131**
	Test technique	152.63	4.181	0.0504
	Subject ability * Test technique	286.82	7.856	**0.0091**
OrdSet	Subject ability	896.66	2.789	0.1053
	Test technique	270.24	0.841	0.3665
	Subject ability * Test technique	215.31	0.669	0.4196

Table 11: ANOVA - Sequential test results (High ability = Block 1)

d. Class cluster characteristics impact

The last factor to study in this section is class cluster characteristics. As we have seen the two clusters are different in terms of the complexity of their statechart and their code characteristics (e.g., real-time behavior). We performed a two-way ANOVA and the results are presented in Table 12. They show a significant impact of cluster characteristics on mutation scores but no interaction effect with test technique. This can be explained as follows: (1) Cruise Control has multithreaded code with real-time behavior and (2) the OrdSet statechart, even though more complex, describes the cluster better than the Cruise Control statechart. As already discussed, the latter provides an incomplete model of the cluster's run-time behavior.

Model RSquare	Factor	Sum of Squares	Parameter estimate	F Ratio	Prob > F
0.499	Cluster Characteristics	6675.04	-14.22	34.05	**<.0001**
	Test technique	511.79	-5.57	2.61	0.1112
	Cluster characteristics * Test technique	66.39	2.01	0.33	0.5627

Table 12: ANOVA - Impact of cluster characteristics and its interaction with test technique on mutation scores

4.2 Combining test techniques to improve fault detection effectiveness

In this section we address four research questions: 3, 4, 5 and 6. Section 4.2.1 analyzes how complementary are the test cases generated based on statecharts (Ts) and those generated based on code (Tc). In Section 4.2.2, we evaluate the impact of combining Ts to Tc on the mutation score. Next, in Section 4.2.3, we investigate the possible interaction effect between the cluster under test and the test technique on the gain in mutation score when combining Ts and Tc. Last, we study in more details the effect of combining Ts and Tc per mutation operator (Section 4.2.4).

When combining test cases, all pairs of drivers (statechart drivers x code drivers) must be taken into consideration to capture the variability among drivers written by different subjects. Having *m* statechart drivers and *n* code drivers implies that *m* x *n* combinations would be considered. However, drivers with low coverage do not represent realistic non-experimental situations with competent developers and a reasonably disciplined process. Those drivers had low coverage due to the combination of three reasons: poor development skills of their authors, lack of compliance to instructions to implement a specific testing strategy, and limited time in the labs. Therefore, we decided to compare only a subset of the drivers by eliminating drivers with low coverage to obtain more realistic analysis results.

To understand how we selected a subset of drivers, one must look at the line charts in Figure 18 (Appendix A), which show the different plots for node, edge, and RTP coverage as well as the corresponding drivers' mutation scores per cluster and per test technique. Drivers are sorted by node coverage for the code test technique and RTP coverage for the statechart test technique. As node coverage in code drivers (respectively RTP coverage in statechart drivers) increases, mutation score increases as well. Based on node coverage for code drivers and RTP coverage for statechart drivers, we identified the following criteria to select drivers for the remainder of this analysis:

- Code drivers with node coverage greater than or equal to 85%.
- Statechart drivers with RTP coverage of 100% for Cruise Control drivers and greater than or equal to 60% for OrdSet drivers.

These thresholds are a compromise between the level of completeness of the test driver and the resulting number of selected drivers that must be large enough to allow analysis. Though it is common practice to seek high statement coverage rates during testing in industrial test environments [45], this rate does not usually reach 100% due to budget and time restrictions, as well as the presence of unreachable code. Thus we chose 85% as a reasonable threshold for the selection of code drivers. As for RTP the decision is more complex as there is no much practice of statechart testing. We chose a 100% threshold for Cruise Control as only two subjects did not cover all RTPs in their drivers. But for OrdSet, very few subjects were able to cover all RTPs (see Table 6). In any case, we suspect that in a typical industrial environment, for a complex statechart with a large number of RTPs, only a subset of them is likely to be selected to fit within available time and effort. Thus we selected a 60% RTP coverage threshold for OrdSet so as to obtain a reasonably large subset of at least half-complete drivers. Table 13 lists the number of

selected and discarded drivers and the resulting total number of driver pairs to consider for the analysis of this section.

	OrdSet	Cruise Control
# Selected code drivers	8	10
# Selected statechart drivers	7	15
# Discarded code drivers	9	6
# Discarded statechart drivers	11	3
Total number of pairs to combine	56	150

Table 13: Drivers selection data for combining test techniques analysis

4.2.1 How complementary are test techniques?

To address question 2, we can first look at the set of faults detected by one type of driver and not the other. Such an analysis needs to be done for all possible pairs of statechart-code drivers and we therefore obtain distributions of joint mutation scores. Figure 12 shows the mutation score distributions and Box plots[2] of all pairs for Ts, Tc, and the differences between their detected fault sets normalized by the total number of faults (F) and expressed in percentages. Appendix B includes all related descriptive statistics|.

Fs–Fc|/|F|% represents the gain in mutation score of code drivers when augmented with test cases from statechart driver. One can note that the number of faults detected only by statechart drivers represent on average a relatively small percentage of all seeded faults in the cluster: 7% for Cruise Control and 12% for OrdSet (see Table 27 in Appendix B). However, this number can sometimes reach considerably larger values (39% in OrdSet) and is probably of practical significance whether to combine techniques.

How much code testing is complementary to statechart testing is captured by |*Fc–Fs*|%. This is probably a more realistic scenario than the one above as in practice one would probably first generate black-box test cases (e.g., based on a statechart), measure code coverage, and complement the test suite to achieve a certain level of coverage. The main reason is that generating large test suites from code coverage analysis only is a highly tedious, time consuming task [33]. The average of |*Fc–Fs*|/|F|% is 14% for Cruise Control and 11% for OrdSet. The maximum increase in mutation scores provided by code drivers was in Cruise Control (33%). This is due to the real-time behavior of this cluster which has kept statechart drivers from reaching high mutation scores.

[2] Box plots show selected quantiles of continuous distributions and extreme values. The ends of the box are the 25th and 75th percentiles, also called the *quartiles*. The line across the middle of the box identifies the median sample value and the means diamond indicates the sample mean and 95% confidence interval. The dashed lines, sometimes called *whiskers*, extend from both ends to the outer-most data point that falls within the distances computed. The bracket along the edge of the box identifies the *shortest half*, which is the densest 50% of the observations.

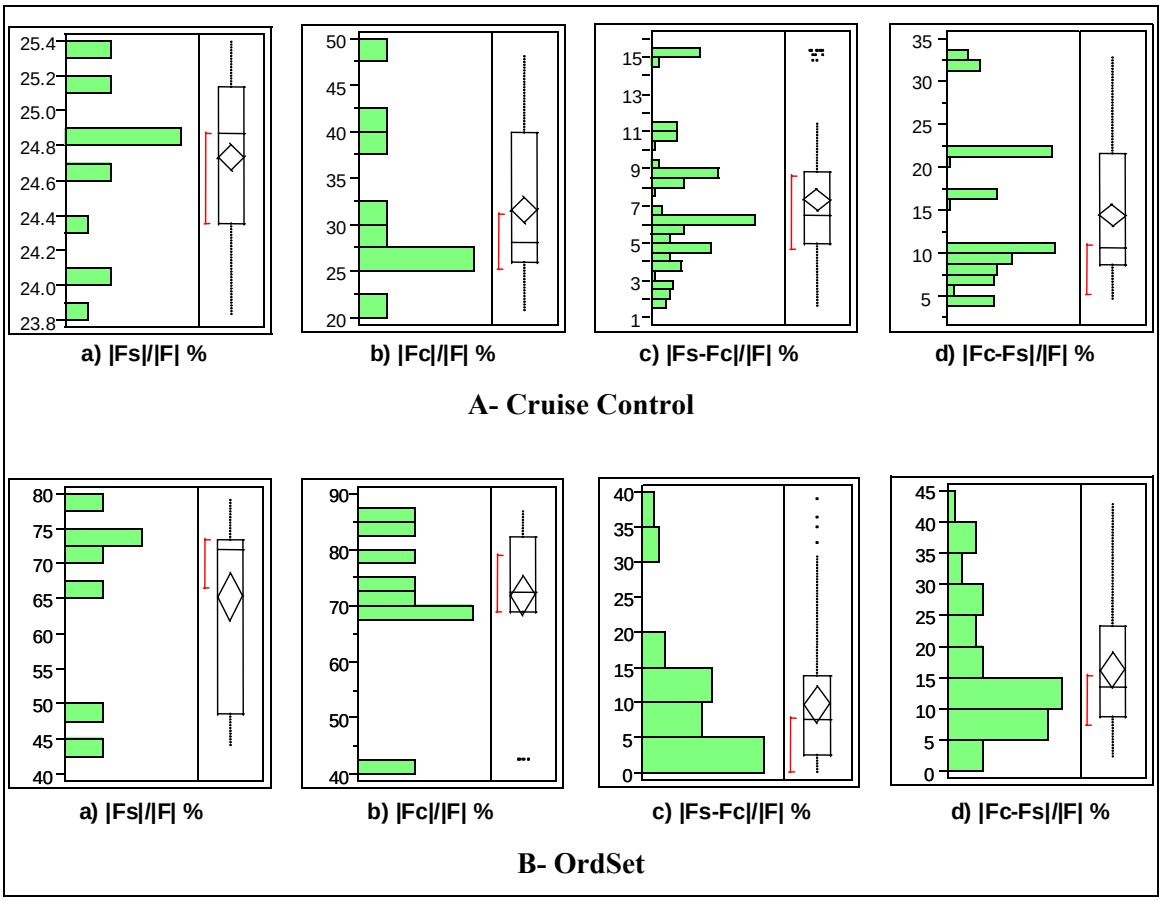


Figure 12: Distributions of mutation scores for detected fault sets

To further investigate the extent to which the two test techniques are complementary, we analyze the normalized intersection between the sets of detected faults by statechart and code drivers |Fs ∩ Fc| / |F|% and the mutation score proportion this intersection represents for each driver (|Fs ∩ Fc| / |Fs| and |Fs ∩ Fc| / |Fc|). Such proportions determine the importance of the contribution of each type of driver to the overall mutation score resulting from combining testing techniques. Distributions of the intersection and its ratios are presented in Figure 13. Related descriptive statistics are presented in Appendix B.

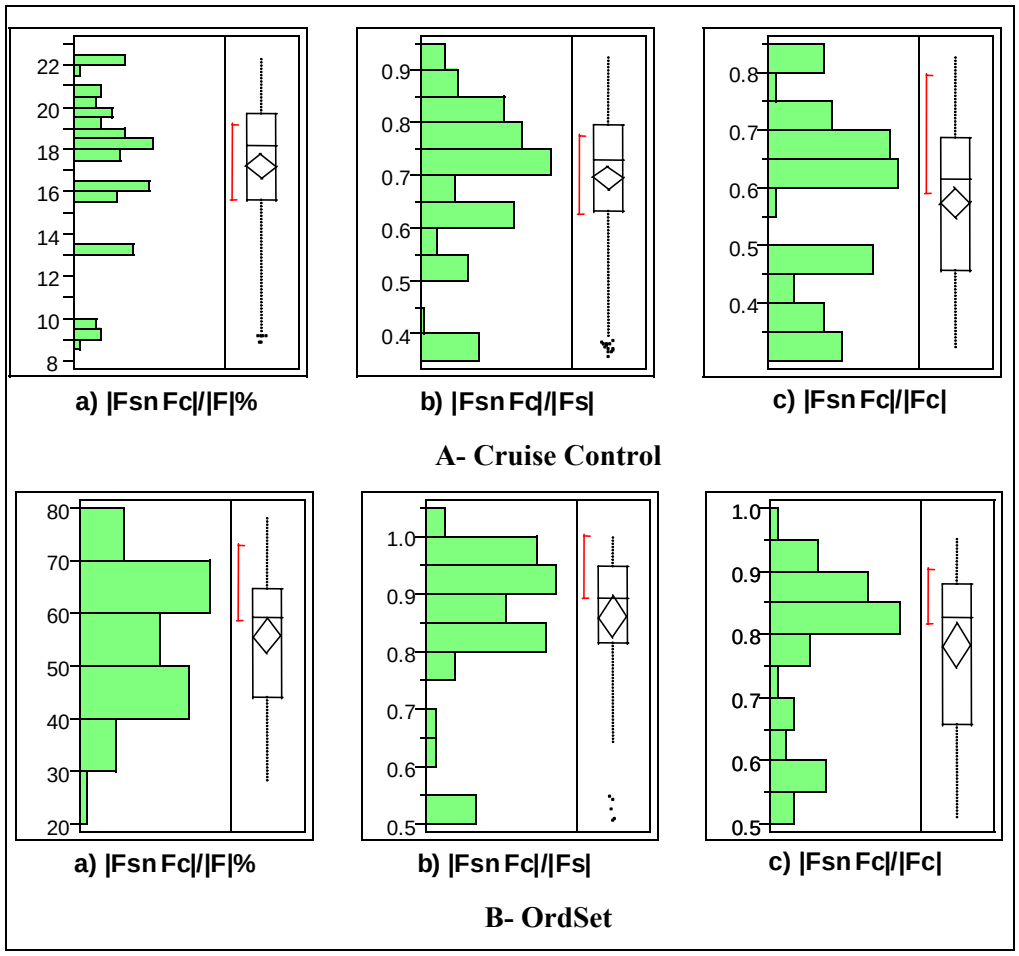


Figure 13: Distribution of intersections

Results show that a practically significant proportion of faults were detected only by one type of drivers. For instance, the average of |Fs ∩ Fc| / |Fs| for Cruise Control is 0.69. This means that on average roughly 30% of the faults detected by statechart drivers are not detected by code drivers. Similarly, on average more than 50% of faults detected by code drivers are not detected by statechart drivers. This further confirms that the two techniques are complementary in terms of fault detection and, as far as statechart testing is concerned, this leads to two questions:

1. Why did the statechart test cases not detect faults detected by code test cases?
2. And can the statechart testing strategy be improved to detect those faults detected only by code drivers?

An attempt to answer these questions is presented in Sections 4.4 and 4.5.

4.2.2 Impact of combining test techniques on fault detection effectiveness

To address research question 4, we need to analyze the significance of the gain in mutation scores when combining statechart and code test cases. We need to test the following null hypothesis: “The fault detection rate when combining statechart testing and code testing is equivalent to that obtained with code testing alone and to that obtained

with statechart testing alone". Two one-tailed *t*-tests for paired samples were performed to compare: (1) the means of mutation scores when including only code test cases and after adding statechart test cases to them, and (2) the means of mutation scores when including only statechart test cases and after adding code test cases to them. Recall each observation corresponds to one pair of code and statechart drivers.

Results of the one tailed *t*-tests are shown in Table 14 and show that the gain in mutation scores, from either code testing or statechart testing alone, is statistically significant when combining test cases from the two testing techniques.

In terms of practical significance, the improvements in mutation scores average between approximately 7% and 10% of all seeded faults across the two clusters when compared to code testing alone. And when compared to statechart testing, the improvement is on average around 13% of all seeded faults.

Cluster	Type of combination	DF	Mean of difference in mutation score	t value	Pr > |t|
Cruise Control	Statechart vs. combination	149	12.69	20.76	**<.0001**
	Code vs. combination	149	7.28	25.00	**<.0001**
OrdSet	Statechart vs. combination	55	13.41	11.75	**<.0001**
	Code vs. combination	55	9.67	7.64	**<.0001**

Table 14: Combining test techniques - Paired *t*-tests results

Figure 14 shows the distributions for the following variables (Table 29 in Appendix B includes the corresponding descriptive statistics):

- |Fs ∪ Fc|/|F|%: represents the mutation score in percentage when combining statechart and code drivers.
- |F – (Fs ∪ Fc)|/|F|%: represents the percentage of faults that remain undetected after combining test cases from statechart and code drivers.
- |Fs ∪ Fc| / |Fs|: represents a ratio measure of the gain in mutation score when combining drivers compared to statechart drivers alone.
- |Fs ∪ Fc| / |Fc|: represents a ratio measure of the gain in mutation score when combining drivers compared to code drivers alone.

It is interesting to note the combined techniques' mutation scores were significantly improved compared to those obtained with each technique individually. For instance, for Cruise Control and OrdSet, the combined techniques' mutation scores represent an average increase of 26% and 17%, respectively, when compared to code testing alone (refer to |Fs ∪ Fc| / |Fc| column in Table 29 - Appendix B). And when comparing with statechart testing alone, the increase reaches an average of 57% and 29% for Cruise Control and OrdSet, respectively. Also, an important result to point is the high mutation scores of the combination achieved for OrdSet (an average of 84%). A more modest result is achieved for Cruise control (an average of 39%) but this can be attributed to the already discussed real-time behavior of this cluster.

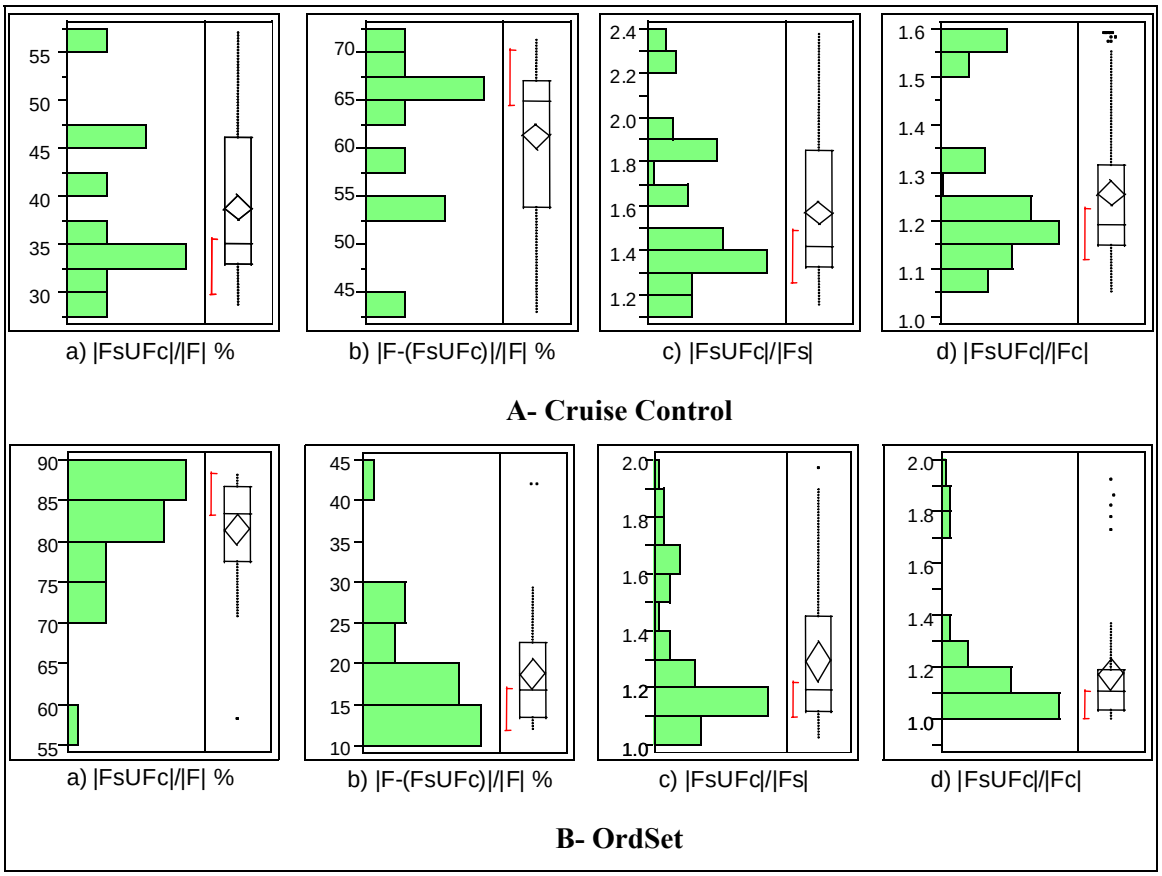


Figure 14: Distribution of mutation scores after combining statechart and code test cases

Another interesting result is the wide range of variability of $|Fs \cup Fc| / |Fc|$: [*min*, *max*] = [5%, 59%] for Cruise Control and [0%, 92%] for OrdSet. For Cruise control, this variability can be attributed to the variability in code driver mutation scores and their detected faults sets. Subjects working with code only, in both clusters, used a wide variety of test suites and applied different levels of precision in their oracles causing an important variability in code driver mutations scores. This is to be expected when one is only driven by coverage but that wasn't the case for subjects working with statecharts as they were instructed to use the RTP test technique with a specified transition tree. This resulted into little variability is the implementation of the technique and therefore statechart drivers for Cruise Control had very close mutation scores and detected almost the same faults. For OrdSet, the variation in mutation score gain can also be attributed to the variability in the number of RTPs covered by the drivers.

4.2.3 Impact of cluster characteristics on combining test cases from both test techniques

We performed two-sample *t*-tests to study the impact of cluster characteristics on the mutation score gain of combining test techniques (1) when compared with test cases from statechart drivers, (2) when compared with test cases from code drivers, and (3) on the percentage of faults remaining undetected. Results are presented in Table 15 and show a statistically significant impact of cluster characteristics on the gain in mutation

scores and the number of faults undetected when combining code and statechart drivers. Results for OrdSet are much more promising than those for Cruise Control due to the already discussed code and statechart properties of the respective clusters.

	DF	Mean OrdSet	Mean Cruise	t value	Pr > |t|
Gain from code mutation score	57.24	9.63	4.51	-4.004	**0.0002**
Gain from statechart mutation score	65.39	16.16	8.85	-5.087	**<.0001**
Undetected faults	79.97	18.64	76.02	57.93	**0.0000**

Table 15: Impact of cluster characteristics and its interaction with test technique on mutation scores

4.2.4 Impact of combining test techniques on fault detection effectiveness per mutation operator

Results discussed in the previous sections show that statechart testing and code testing are complementary overall. Both contribute significantly to fault detection rates. In this section we further investigate their impact on fault detection effectiveness at the mutation operator level.

Table 16 lists the total number of mutants created per mutation operator (|F|), the number of mutants killed only by statechart drivers (|Fs-Fc|), those only killed by code drivers (|Fc-Fs|) and the total number of mutants killed when combining test cases from both drivers (|Fs∪Fc|). Recall that the number of mutants for a mutation operator changes with the cluster under test as this number is related to the characteristics of the code. Related descriptive statistics are presented in Appendix C.

Note that for better readability, only a minimal subset of variables was listed in Table 16. Other variables can be derived as follows:

- Number of mutants killed by statechart drivers: $|Fs| = |Fs \cup Fc| - |Fc - Fs|$,
- Number of mutants killed by code drivers: $|Fc| = |Fs \cup Fc| - |Fs - Fc|$
- Number of mutants killed by both types of drivers: $|Fs \cap Fc| = |Fs \cup Fc| - (|Fc - Fc| + |Fs - Fc|)$
- Number of live mutants: $|F| - |Fs \cup Fc|$

When combining the two test techniques, most faults seeded in OrdSet (mutants) were detected. Most undetected faults do not affect the results of the faulty method and therefore correspond to equivalent mutants. For example, a seeded right-shortcut on a parameter *D* (*D*++) in an assignment statement is computed after evaluating the statement. When *D* is not used afterwards in the faulty method, the error does not propagate and does not affect the output of the method. For JSD mutants, the faults cannot be detected as there is no code in OrdSet that changes the value of static attributes. For Cruise Control, JSI and JSD mutants (inserting and deleting static keyword) were not killed as drivers created only one instance of the car. JID mutants were not killed either (attribute initialization deleted) as attributes were set to default values, e.g., 0 for integer attributes. However, high numbers of other types of mutants were killed and the

remaining live mutants were mostly due to the difficulty of devising precise oracles with exact values for class attributes.

Mutation operator	OrdSet				Cruise Control			
	$\|F\|$	$\|Fc-Fs\|$	$\|Fs-Fc\|$	$\|Fs \cup Fc\|$	$\|F\|$	$\|Fc-Fs\|$	$\|Fs-Fc\|$	$\|Fs \cup Fc\|$
JSI	5	1 (20%)	0	4 (80%)	11	0	0	0
JSD	3	0	0	0	11	0	0	0
JID					9	0	0	0
JDC					1	0	0	1 (100%)
EAM	4	0	0	4 (100%)	5	3 (60%)	0	4 (80%)
AORB	90	6 (6.7%)	4 (4.4%)	90 (100%)	32	28 (87.5%)	0	28 (87.5%)
AORS	8	0	1 (12.5%)	8 (100%)				
AOIU	48	0	5 (10.4%)	48 (100%)	32	15 (47%)	0	26 (81%)
AOIS	297	16 (5.4%)	2 (0.7%)	236(79.5%)	144	55 (38%)	1(0.7%)	94 (65%)
AODU	4	0	0	4 (100%)				
ROR	47	3 (6.4%)	0	40 (85%)	79	25 (32%)	0	48 (60.8%)
COR	6	0	1 (16.7%)	6 (100%)				
COD	1	0	0	1 (100%)				
COI	4	0	0	4 (100%)	1	0	0	1 (100%)
LOI	107	3 (2.8%)	3 (2.8%)	106 (99%)	51	10 (19.6%)	0	37 (72.5%)
ASRS	0	0	0	0	12	12 (100%)	0	12 (100%)

Table 16: Count of detected and live mutants per mutation operator

Results in Table 16 also help identify mutation operators for which a test technique is a better detector than the other (research question 6). A first trend to notice is that the cluster under test seems to be an important factor that impacts mutation operators for which statechart drivers are good detectors. For instance, out of the 90 AORB mutants created for OrdSet, 84 of them have been killed by statechart drivers. However, none of the 32 AORB mutants created for Cruise control has been killed by statechart drivers ($|Fs- Fc| = |Fs \cup Fc|$. The AORB mutation operator replaces a binary operator such as the addition operator with another binary operator. In Cruise Control, such faults are seeded in the algorithm computing class attributes when the car is running (car speed, throttle …). In order to detect such fault, a precise oracle is needed. However, it is extremely hard to know at some point in time what would be the value of car speed for example. Such value depends on many factors: execution time, processor speed, and number of running processes on CPU. Therefore, oracles for Cruise Control cannot be very precise; an attribute value can be only checked against an interval. However, for OrdSet, the characteristics of the class allow for precise oracles. At any point in time, one can check class attribute values against exact expected values. Also for AOIU mutants, the trend is inconsistent in the two tested clusters. For Cruise Control, more faults were detected by code drivers, as opposed to OrdSet where more faults were detected exclusively by statechart drivers. This result can be also attributed to the precision of oracles used in the two clusters.

For Cruise Control, no mutant of type ASRS was killed by statechart drivers, such a fault would alter the value of a numerical attribute. This again can be attributed to the fact that statechart drivers use state invariants as oracles and state invariants for Cruise Control are not very precise as described in the discussion above. It may not be possible to define an exact value for some class attributes, even in a method postcondition. Certain properties are just inherently hard to express with contracts. This is particularly true for contracts of computationally intensive methods for example. Writing such an oracle would consist in simply rewriting all the code statements in a language such as OCL and might end up to be more complex than the code it is supposed to describe. We expect that covering an activity diagram, such as the one presented in Figure 6 describing the algorithm that manages the Cruise Control class attributes, would help identify more precise oracles by narrowing down ranges of values to check and therefore would help to detect AORB and ASRS faults.

For the other mutation operators, coherent trends are seen for both clusters. Only notice that OrdSet drivers had higher mutation scores than Cruise Control drivers. This is related to the different code and statechart properties of the two clusters that have been already discussed above.

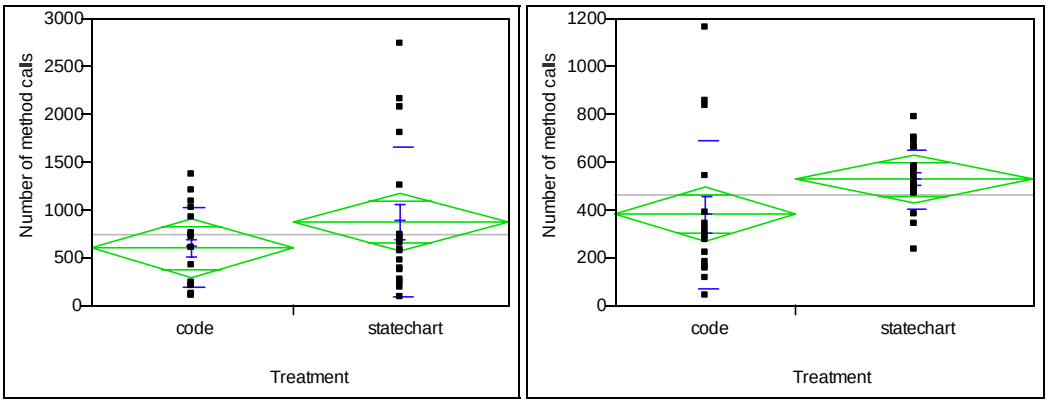
From the above discussion, we cannot conclude that statechart drivers are better fault detectors than code drivers for any of mutation operators. However, code drivers can be better fault detectors than statechart drivers depending on cluster characteristics. It is therefore important to investigate the possibility of improving statechart testing to address its main weaknesses. Section 4.4 will investigate in a thorough and systematic manner the main causes for statechart drivers to fail detecting faults.

4.3 Comparing the cost-effectiveness of test techniques

In this section we attempt to answer research questions 7 and 8 by studying the difference in terms of cost and cost-effectiveness between statechart testing and code testing.

4.3.1 Cost Analysis

Recall from section 3.3.2 that we assume the cost of a driver (a test set) to be proportional to the number of method calls to the classes under test. Figure 15 shows the distribution of drivers' cost per cluster and per test technique. Related descriptive statistics are reported in Table 17. We can see that in both clusters, statechart drivers tend to have a higher average cost. Two-sample *t*-tests were performed to obtain statistical evidence about the impact of test technique on the cost of drivers (research question 7). Results reported in Table 18 show that the difference between the two test techniques is not significant. However, a non-parametric Wilcoxon test shows a significant difference in cost between test technique for Cruise Control (Prob > |Z| was evaluated to 0.0211). Since a *t*-test tends to be conservative when the observations' distribution is not normal, we will tend to rather trust the Wilcoxon test results, thus concluding that statechart testing is significantly more expensive than code testing for Cruise Control.



a) OrdSet b) Cruise Control

Figure 15: Cost comparison of statechart and code drivers

Cluster	Treatment	Min	Max	Mean	StdDev	Range
OrdSet	Statechart	73	2732	875.55	781.48	2659
	Code	84	1350	600.64	409.26	1266
Cruise Control	Statechart	230	683	502.17	114.60	453
	Code	39	1155	382.43	310.08	1116

Table 17: Cost descriptive statistics

Cluster	DF	Mean Code	Mean Statechart	t value	Pr > |t|
Cruise Control	18.81	382.43	502.17	-1.453	0.1624
OrdSet	25.98	600.64	875.55	-1.313	0.2004

Table 18: *t*-test results for cost analysis

4.3.2 Cost-effectiveness analysis

We measure the cost-effectiveness of a driver as the ratio of its mutation score to its cost. We performed one-tailed *t*-tests to assess the impact of the test technique on the cost effectiveness of test drivers (research question 8). Results are reported in Table 19.

Code testing is found to be significantly more cost-effective than statechart testing. However, from the previous section we have seen that code and statechart testing tend to be complementary. Also, we have seen that statechart testing tends to be less affected by the skills and ability of testers as it is better defined and more systematic. Last, statechart testing can be planned early on before any code is available. Therefore, despite a different in cost-effectiveness, these results do not suggest in any way that code testing should be the preferred choice over statechart testing.

Cluster	DF	Mean Code	Mean Statechart	t value	Pr > t
Cruise Control	15.40	0.12	0.05	2.918	**0.0052**
OrdSet	22.82	0.15	0.09	-1.935	**0.0307**

Table 19: *t*-test results for cost effectiveneness

4.3.3 Investigating the variation is cost-effectiveness

Figure 16 shows scatter plots for each cluster and test technique: mutation score as function of driver cost. We expect a positive, monotonic relationship between the two variables and we would expect low dispersion if the cost-effectiveness were similar across drivers. However, we can clearly see a great deal of dispersion on the plots for code testing, thus suggesting a lot of the variation in mutation score is not explained by the size of the driver.

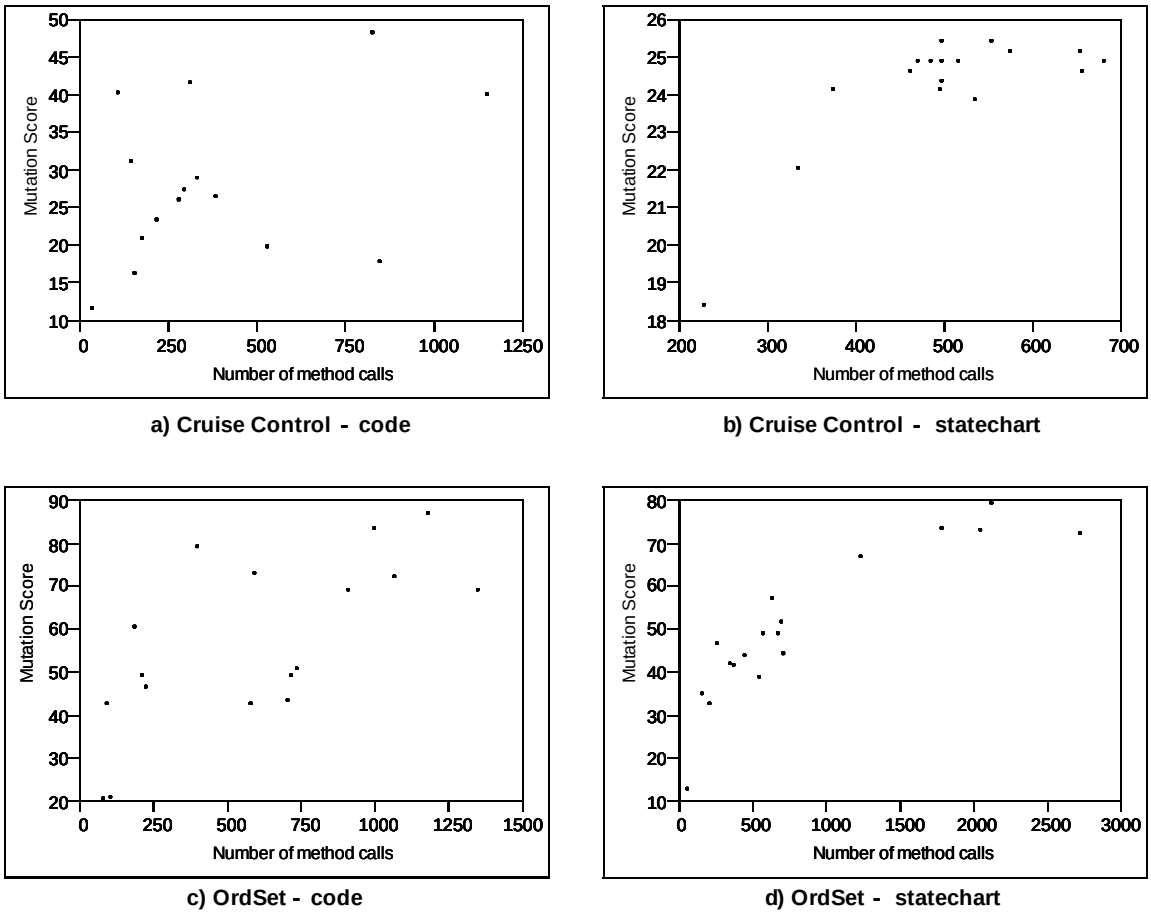


Figure 16: Mutation Score - Cost scatter plots

We performed a qualitative cost analysis of drivers to investigate the reasons for such variability. The results of this analysis are summarized in Table 20. The main cause for cost-effectiveness variability in code drivers is a frequent redundancy in test cases, writing multiple test cases that partially cover the same nodes/edges. Redundancy is however controlled in the case of statechart testing as the test cases are precisely specified by a test strategy (RTP) and therefore this leads to less code coverage redundancy. Another source for variability in cost effectiveness is the ineffective use of available public methods to implement certain functionalities. For example, in OrdSet, a set can be created with two constructors, one creates an empty set and another creates a set with content from an array of integers. Some subjects did not use the second type of constructors to create a non-empty set. Instead, they created an empty set and iteratively added elements to it with the "add one element" method. This increased the number of

called methods in their drivers considerably. Similarly, in Cruise Control, the inefficient use of command calls to accelerate or to brake to reach a certain speed unnecessarily lead to considerable increase in driver cost. For instance, calling the accelerate command twice in a sequence produces the same behavior as calling it any number of times.

Cruise control – code drivers
- High numbers of test cases, which are not necessarily good detectors of faults, i.e. redundancy in terms of node/edge coverage. - High precision in oracles, i.e. verifying values of all or most class attributes leading to high numbers of method calls. - Long paths of commands, for example 20 consecutive times of "accelerating" or 50 times "braking", this is not necessarily helpful in detecting faults as the maximum speed or maximum brake value can be reached in less commands. Also when accelerating without waiting for speed to increase only two accelerating commands are equivalent to any number of accelerating commands above two.
Cruise control – statechart drivers
- Verifying oracles only after the last transition in a round-trip path (causing high variation) - Wrong implementation of round trip paths, i.e. implementing 16 RTPs instead of 12.
OrdSet – code drivers
- Creating of a set with content as an empty set and adding contents one by one afterwards by calling the method "add"; note that the creation of this kind of set can be created with a single method call, i.e. the constructor that accepts an array of integers as a parameter. - High number of test cases with less precise oracles. Note that even when the number of test cases increases, the mutation score may not increase as additional code coverage may not be proportional to the number of test cases. - High precision in oracles (every oracle checks for all attributes values). - No apparent use of a systematic testing strategy leading to high number of test cases.
OrdSet – statechart drivers
- Testing unnecessarily the values of test cases settings; for example, if a test case require an empty set as a parameter, the test case would include a test for the emptiness of the provided set. - Verifying oracles only after the last transition in a round-trip path (causing high variation). - Wrong implementation of state invariant assertions causing lower driver sizes; for example the state "Empty" is tested with the "isEmpty" method instead of testing the invariant of state Empty which includes the verification of additional attributes. - Creating of a set as an empty set and iteratively adding content by calling the method "add"; note this can be created with only one method call, i.e. the constructor that accepts an array of integers as a parameter. - Number of test cases (not all students were able to implement all RTPs in the transition tree).

Table 20: Causes for variation in driver cost effectiveness

4.4 Qualitative analysis of live mutants

We discussed in Section 4.1 the fault detection effectiveness of statechart drivers and we have seen that large numbers of mutants in Cruise control were not detected and fewer numbers of mutants were not detected in OrdSet. To better understand why certain

faults are difficult to detect by statechart drivers, we performed a qualitative analysis to identify what execution conditions would be required to detect faults and whether these conditions were likely to be fulfilled by statechart testing. This also helps us identify how to improve statechart testing in order to increase its fault detection effectiveness. This analysis was systematic and included the following three steps:

1. Running perl scripts to identify the following disjunctive sets of faults: (1) F-(Fs ∪ Fc) = set of faults not detected by any driver, (2) Fc-Fs = set of faults detected only by code drivers, (3) Fs-Fc = set of faults detected only by statechart drivers, and (4) Fs∪Fc = set of faults detected by both types of drivers.
2. Identifying the reasons for not detecting faults with a focus on F-(Fs ∪ Fc) and Fc-Fs. This was done by executing the corresponding mutants and generating traces of execution. If the fault does not affect the output, the trace can then help us identify the reason. And if the fault does indeed affect the output, the trace then helps us understand why the oracle did not detect any failure. An example of such faults is one created by seeding a fault in the method "*resizeArray*" of the *OrdSet* class as shown below:

```
private void resizeArray() {
     int new_size = _set_size + min_set_size;
       if (new_size <=max_set_size &&
       _resized_times<max_accepted_resizes) {
         int[] _new_set = new int[new_size];
         for (int k = 0; k < _last + 1; k++) {
             _new_set[k] = _set[k];
         }
         _set_size = new_size;
         _set = _new_set;
         _resized_times++;
     } else {
         _overflow = true;
     }
 }
```

Recall that the method *resizeArray* is called whenever an element is to be added to a full set. To cause a resize, the element to be added should not be already in the set. A resize can occur if two conditions are true: (1) the resized set size does not exceed the maximum set size (a constant), and (2) the number of resizes done on the set does not exceed the maximum resizes allowed (a constant). The error was seeded by replacing the index *k* highlighted above in the code by *k*++. The fault gets executed if a new element is to be added to a full set in which the conditions for a resize are fulfilled. When the fault is executed, the set is resized as expected, but with wrong content (some elements would be replaced by zero). In order to detect the fault, a verification of the set content is needed. This can be done by verifying the class invariant or the `resizeArray` postcondition in the oracle. An example of a test case that causes a failure if this fault is executed is to create an ordered set with content {1, 2, 3, 6} then to add the element "4" to the set. The result one gets is {0, 2, 0, 4, 6} instead of {1, 2, 3, 4, 6}.

3. Classifying undetected faults according to the categories listed in Table 21 below.

Cruise Control		**Total**
1	Mutants corresponding to inserting or deleting "static" keyword were not detected by any driver as only one instance of the cruise system (Controller and Carsimulator) was running at a time in all drivers.	20
2	Deleting initial values of attributes were not detected as default values are equal to the initial values deleted. Example: an integer attribute that is supposed to be initialized to 0.	9
3	Faults that affect the algorithm that manages the speed and throttle. For example: (1) a fault that causes air resistance to become positive instead of negative, (2) a fault that causes the throttle to increase instead of decay with time when the accelerator is not pressed. These faults cause wrong values of class attributes such as speed. Such fault requires a very precise oracle in order to be detected.	158
4	Fault seeded in dead code cannot be detected.	1
5	Faults that do not cause a behavioral change in methods. They are just artifacts of the way we seeded faults using mutation. It is common to see so-called "equivalent" mutants when using mutation operators to seed faults [4, 39, 43].	24
6	Faults that can be detected only with a specific test case, mainly requiring repeating a call of the same command in a test case (path) and/or causing the system to sleep to allow values of speed, throttle, or distance to change over time. Such faults may be detected by code drivers if a specific path is implemented, and they cannot be detected with statechart drivers because the testing technique used does not allow the repeating of commands more than once in a round trip path.	10
8	Faults that can be detected with test cases for some paths that are not covered in the statechart, i.e. unspecified self-transitions or sneak paths (example, in state inactive, the event "on" is not represented with a self-transition). Subjects using statecharts implemented only paths in the transition tree in their driver and therefore do not account for sneak paths. Subjects working with code did not think of covering sneak paths as well probably because they correspond to omitted "else" statements in "if" statements.	14
9	Drivers did not execute the cluster during a long enough period of time with appropriate conditions to trigger specific code/behavior.	41
OrdSet		
1	Faults that delete or add a static keyword. Static attributes are used in the cluster as constants. There is no code that affects static attributes values. Thus these faults cannot be detected.	4
2	Faults that cause no behavioral change. These again, are equivalent mutants and are artifact of our fault seeding procedure.	58
3	Faults that cause an infinite loop. No oracle checks whether execution takes too long.	49
4	Faults that cause wrong set content and/or actual set size. These faults can be detected with methods' post conditions and class invariant assertions in oracles.	41
5	Faults that can only be detected with some specific parameters values in test cases, i.e. a specific set size or a specific content.	8

Table 21: Classification of causes for not detecting faults

By looking at Table 21 we notice that the identified categories of undetected faults are related to the characteristics of each cluster. Once again, how well a technique is

going to detect faults depends to a great extent on the characteristics of the software under test.

For Cruise Control, most undetected faults are from the third category (158 faults). As previously discussed, the large number in this category can be attributed to the fact that the algorithm that determines the values of class attributes is not described by the statechart. These faults result in wrong values assigned to class attributes. In order to detect these faults, not only precise oracles should be implemented but also test driver execution time should be increased (add wait time) to allow for substantial changes in class attributes values. Another important category of undetected faults is the one associated with the real-time behavior of the software under test (category 9). These faults may be detected if more time is allowed for drivers' execution, i.e. using the "sleep" method from the Thread class.

For OrdSet, most undetected faults are from the second category. These "faults" correspond to equivalent mutants and cause no behavioral change in system: they should therefore be ignored in our analysis. Another important category of undetected faults is the third category: these faults cause infinite loops. They can only be detected if drivers are designed to detect abnormal execution times. Faults in category 4, which are frequent, can be addressed by implementing contract assertions in oracles.

From the results above we note that equivalent mutants correspond only to 9% and 36% of undetected faults and 6% and 9% of total seeded faults for Cruise Control and OrdSet, respectively. Therefore it confirms that a heuristic, sometimes used in testing empirical studies [1, 4, 15, 19, 39, 43], that consider all faults undetected by any driver as equivalent mutants and eliminate them from the total set of mutants cannot be applied in our case. However, we can adjust based on our qualitative analysis the mutation scores of all the test drivers of our experiment.

Let M_b and M_a be the mutation score of a driver before and after removing equivalent mutants respectively, N_e the number of equivalent mutants and F the total number of mutants. Then Ma can be computed as follows:

$M_a = (M_b*F/100 - N_e)*100/F$

We computed M_a for all drivers and both clusters. Results are presented in Table 22. All the mutation scores dropped but after re-running the various analyses we performed, these changes turned out not to make any difference in terms of the conclusions we have drawn in the previous sections.

		Mean of drivers mutation scores		
		statechart drivers	**code drivers**	**statechart + code drivers**
Cruise Control	**With equivalent mutants**	24.47	27.69	35.86
	Without equivalent mutants	16.73	20.28	31.61
OrdSet	**With equivalent mutants**	50.27	56.15	71.41
	Without equivalent mutants	45.17	51.65	68.48

Table 22: Equivalent mutants' impact on mutation scores

4.5 Proposition of improvements to statechart testing

The qualitative analysis of live mutants (Section 4.4) helped us address research question 9 and identify additions to our statechart testing strategy in order to improve the fault detection ratio of statechart drivers. The proposed improvements as listed below are based on the two tested clusters but they can be easily generalized to many control and complex data structure classes:

1. Use activity diagrams to model algorithms with high control flow complexity. They will help ensure minimal edge coverage for the most complex methods and would also help describing the impact of the "time" factor on the different class attributes in a real-time system. The coverage criterion would be to cover all paths in the activity diagram. If a loop exists, then typical loop coverage heuristics should be applied [8].
2. Include contract assertions (class invariant and methods post conditions) in oracles in addition to state invariants.
3. Complement the round-trip path technique with sneak path testing. This has been recommended by Binder [8] and our data clearly confirm this is required. Including explicit self-transitions in statecharts to account for sneak paths within RTPs would not be a convenient alternative as statechart would become unnecessarily complex.

An example of an activity diagram for a running car in Cruise control is provided in Figure 17. Such an activity diagram helps identifying test cases that cannot be otherwise identified based on the statechart and using the Round-trip path technique. For instance, to perform boundary testing in the case of Cruise Control, we need to test the case where a car is running at maximum speed. However, to reach the maximum speed, the car should accelerate for some time as speed is a function of time (changes every 200 ms). This corresponds to executing the highlighted path in Figure 17 a number of times.

Therefore, test driver execution time should be extended to allow speed to increase. Recall that the time factor does not show in the statechart. Also, one can note from the activity diagram that the speed is a function of the car throttle. The throttle value is increased by a fixed amount for every accelerate command (refer to method "accelerate" postcondition in Appendix E.1.3) and it decays with time as can be seen in the shadowed part in Figure 17. Therefore, in order to continue to bring speed to its maximum value, the command "accelerate" should be repeated a number of times. Implementing such a test case means that the same transition would be executed a number of times. This cannot be implemented with the RTP test technique where a transition cannot be executed in a path more than once. Covering all paths in the activity diagram implies that some paths would be traversed a number of times in order to be able to cover other paths. For example, to cover the path in the activity diagram where a speed is greater than maximum speed, the car speed should first reach maximum speed. In the discussion above we explained that in order to reach the maximum speed the highlighted path in Figure 17 should be executed a number of times.

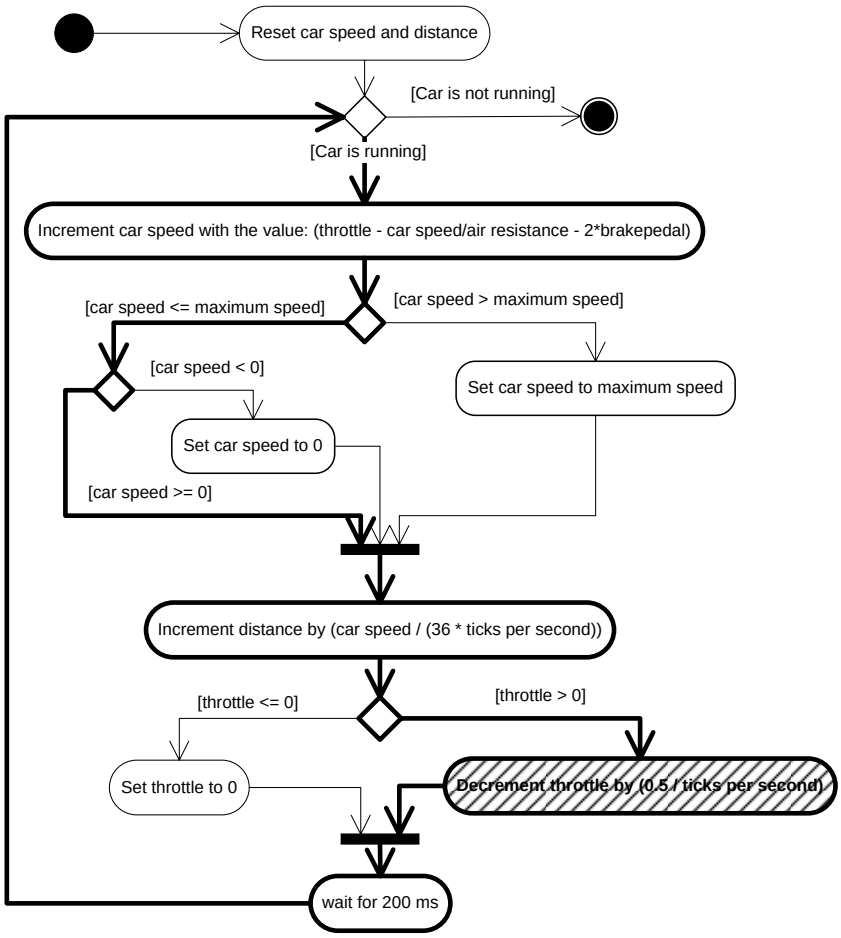


Figure 17: Running car activity diagram with highlighted path

Applying the second improvement to OrdSet would likely help to detect faults such as the AOIS faults. AOIS faults are seeded by inserting arithmetic shortcuts (++, --). They cause wrong content or order of elements in the set. Such faults can be detected if contracts are implemented in oracles to verify the content of a set and the order of its elements.

For Cruise Control, applying the third improvement by including explicit self-transitions with no actions to the statechart would increase the total number of RTPs in transition tree from 12 to 25. But any faults, such as AOIS faults that insert arithmetic shortcuts to state attributes causing a wrong output state, would be detected if sneak paths were tested.

We implemented the above proposed improvements to the model to verify their impact on fault detection effectiveness for statechart drivers. We decided to implement the test drivers and not use the ones implemented by students. The main motivation was to accurately assess the impact of refining our test strategy by implementing fully correct and complete statechart drivers.

For each cluster, we first implemented one base drivers with test cases corresponding to RTPs in the transition trees provided to subjects during the experiment

and using state invariants as oracles. We then augmented them with the suggested improvements.

Only the second proposed improvement applies to OrdSet. A second test driver is created for OrdSet by augmenting test cases in the base driver with contract assertions in oracles. A comparison of the mutation scores, code coverage, and size (cost) of the two drivers is presented in Table 23. Results show an important increase in the number of killed mutants from 494 to 536 mutants (79% to 86%). As a comparison, the highest mutation score in code drivers is 541 mutants or 87% which is very close to the mutation score of the new statechart driver after implementing contract assertions. Node and edge coverage did not change much as no new test cases were added to the driver: changes were only made to oracles that called additional getter methods. Regarding driver cost, the increase is attributed to method calls in contracts where getters are called to verify attribute values. Even with an increase in cost of 15%, the increase in mutation score (8%) is of practical interest.

	Mutation score	Node Coverage	Edge Coverage	Cost (# of method calls)
RTPs + state invariants	79.17	76.09	72.61	2136
+ contract assertions in oracles	85.89 (+8%)	76.81	73.88	2490 (+15%)

Table 23: Improvements to the model impact on OrdSet drivers

Table 24 reports on the changes in the numbers of killed mutants per mutation operator after adding contract assertions to the OrdSet test driver. The mutation operators not showing in the table had no change in number of killed mutants. Note the important improvement in AOIS detection rate (from 209 to 235 killed mutants). This confirms our expectations earlier in this section that AOIS faults (insertion of shortcuts), which cause wrong content and element orders, would be better detected when contract assertions are added to oracles.

	AORB	AORS	AOIU	AOIS	ROR	LOI
RTPs + state invariants	82	7	44	209	37	95
+ contract assertions in oracles	87 (+5.5%)	8 (+12.5%)	45 (+2%)	235 (+9%)	40 (+6%)	101 (+6%)

Table 24: Comparison of number of killed mutants per mutation operator - OrdSet

All three proposed improvements apply to Cruise control. Therefore drivers have been created to implement the three additions to base statechart driver. We first considered each improvement separately to study its isolated effect on fault detection. Next, we proceeded to combine test cases from the different proposed improvements to determine their combined effect.

Mutation score, code coverage, and driver cost data are presented in Table 25. The first proposed improvement aim to cover all paths in the activity diagram and resulted in considerable mutation score improvement (from 25% to 42%). In fact, covering the activity diagram (one test case that covered all paths in the activity diagram) showed the most important impact on the mutation score.

	Mutation score	Node Coverage	Edge Coverage	Cost (# of method calls)
RTPs + state invariants	25.39	78.16	62.05	515
+ activity diagram coverage	42.23 (+17%)	86.18	76.60	543
+ sneak paths coverage	31.61 (+6%)	85.37	76.78	903
+ contract assertions in oracles	31.87 (+6.5%)	79.83	64.53	565
+ contract assertions & sneak paths coverage	39.38 (+14%)	82.58	74.96	1539
+ contract assertions & activity diagram coverage	50.78 (+25%)	82.45	73.62	620
+ sneak paths & activity diagram coverage	47.15 (+22%)	88.07	82.24	941
+ all three proposed improvements coverage	58.29 (+33%)	88.07	82.24	2297

Table 25: Improvements to the model's impact on cruise control drivers mutation scores

Performing all three improvements in the statechart driver caused an increase in the mutation score from 25% to 58% (Table 25). As a comparison, the highest mutation score of code drivers was 48%. Another interesting point to mention is that covering sneak paths had the highest impact on edge coverage. As for cost, sneak path coverage was the most expensive in terms of increase in number of method calls. This can be attributed to the large number of additional RTPs in the new transition tree. The least expensive proposed improvement was the coverage of the activity diagram. This was the most cost effective among the three proposed improvements and this can be easily explained as most undetected faults are real-time dependent (refer to Section 4.4).

Table 26 reports on the changes in the numbers of killed mutants per mutation operator after adding the improvements to the Cruise control base driver. The mutation operators not showing in the table showed no change in number of killed mutants. As expected from our discussion in Section 4.2.4, the detection rates of AORB and ASRS were improved when covering the activity diagram. Also there was an important improvement to the AOIS detection rate especially when covering all the paths in the activity diagram.

	AORB	AOIU	AOIS	ROR	LOI	ASRS
RTPs + state invariants	0	9	32	23	27	0
+ contract assertions in oracles	0	9	54 (+15%)	26 (+4%)	27	8 (+67%)
+ sneak paths coverage	0	9	48 (+11%)	29 (+8%)	29 (+4%)	0
+ activity diagram coverage	5 (+16%)	14 (+16%)	66 (+24%)	31 (+10%)	32 (+10%)	8 (+67%)
+ contract assertions & sneak paths coverage	0	11 (+6%)	59 (+19%)	29 (+8%)	29 (+4%)	8 (+67%)
+ contract assertions & activity diagram coverage	5 (+16%)	15 (+19%)	72 (+28%)	35 (+15%)	32 (+10%)	12 (+100%)
+ sneak paths & activity diagram coverage	5 (+16%)	14 (+16%)	78 (+32%)	35 (+15%)	34 (+14%)	8 (+67%)
+ all three proposed improvements coverage	5 (+16%)	15 (+19%)	82 (+35%)	36 (+16%)	34 (+14%)	12 (+100%)

Table 26: Comparison of number of killed mutants per mutation operator - Cruise Control

4.6 Discussion of Results

Despite being proposed as an efficient strategy for testing state-dependent class clusters [9, 10, 38], the statechart-based testing of source code does not appear more effective at detecting faults than simple structural testing. This is at least the case in the context of our experiment where the time allocated is limited and the same for both test strategies. However, our results also show clearly that statechart and structural testing are complementary in terms of the faults they detect and they should somehow be used together. Since statechart testing can be planned and prepared early before code is ready, and because structural testing is a difficult and tedious task requiring control flow analysis, it is probably better to recommend that statechart testing be used first and then complemented through coverage analysis to reach acceptable levels of code coverage. This is also consistent with the more general recommendation by Marick [33] on black-box and white-box testing.

Our results also show that the fault detection effectiveness of statechart-based testing varies to a large extent depending on how precisely the statechart describes the behavior of the software under test. In our experiment, for a Cruise Control class cluster which behavior is strongly driven by time-related properties, the statechart was only a rough model to base testing on. Only when complemented with test cases covering an activity diagram describing the computation of time dependent class attributes we were able to detect most seeded faults. Another source of variation in fault detection effectiveness is related to the level of precision of the statecharts and its related model elements (contracts, class diagram, and state invariants) which inherently depends on the nature of the software being modeled. For example, many real-time properties and complex computations are typically not represented in statecharts. Furthermore, certain operations cannot be precisely modeled with contracts: this is the case of complex computations. Considering that we are only scratching the surface of the problem here, it is very probable that to be practical and reliable, statechart testing must be complemented

with other testing strategies and that we need to provide precise guidelines regarding its usage and integration with other testing techniques. However, much more experimental research needs to be done to devise a complete strategy.

Statechart driven testing can however be made much more effective by ensuring that illegal (often implicit) events be tested in every state and by implementing precise oracles based on class invariants and contract assertions, in addition to the standard state invariants. As previously discussed in another context [10], we see here that the test strategy is only part of the picture when investigating fault detection effectiveness. The strategy followed to implement oracles is at least as important.

Code testing shows much more variation in effectiveness since it relies much more on the testers' ability and skills as it offers much less guidance than statechart testing. Our results show that, despite being similarly effective overall at detecting faults as code testing, statechart testing is more effective for less skilled testers. In other words, it may be recommended if testers have little experience and weak programming skills and if a choice between the two test strategies has to be made. From a more general standpoint, these results suggest that human factors play an important role in the results of empirical test research. This is probably something that should be more often accounted for in testing research.

5 CONCLUSIONS

This paper investigates the cost-effectiveness of statechart testing of class clusters with state-driven behavior. This is of practical importance as state-driven testing has been often recommended for complex class clusters in literature [9, 10, 14, 16, 17, 38, 42]. As a baseline of comparison we compare statechart-driven testing with the common practice of using coverage analysis of code to drive the development of test suites. Furthermore we investigate whether the two strategies are complementary in detecting faults. We then investigate the factors that may affect how effective these strategies are in practice.

To address these issues, this paper presents the results of a controlled experiment performed in a university environment with senior, carefully trained students. Results show, in a context where time is limited, that a well-known strategy for statechart testing (referred to as the W-method [18] or round-trip path testing [8]) is not more effective at detecting faults than testing driven by code coverage analysis. Furthermore, whether statechart testing is preferable to coverage driven testing seems also to depend on the programming and testing skills of the tester. However, the two test strategies also seem to be complementary in terms of the faults they detect and this suggests that they should probably be used together, as opposed to being alternatives. Because statecharts are available before code is available and because testing based on code coverage analysis is notoriously tedious and time consuming, it is probably wise to first test class clusters based on statecharts and then complement test suites based on coverage analysis.

The results also suggest that the effectiveness of statechart testing strongly depends on the nature of the software under test, the statechart model itself, but also the way test oracles are implemented. As a result, it seems clear that much more research is needed in order to provide clear and precise guidelines to testers as to when to use statechart testing

and how to integrate it with other test strategies. For example, this paper shows cases where it is advisable to complement the statechart with activity diagrams describing the control flow of certain operations (e.g., modeling the way time-dependent class attributes are updated) and then trying to cover the paths in such activity diagrams to complement statechart testing. Our results also show the large extent of the impact of using precise oracles based on contract assertions and class invariants, as well as the necessity to test for illegal and often implicit events in statechart.

Though our two class clusters involved in our experiment are small, they are representative of two typical types of clusters: complex data structures and control classes in a control system. Though our experiment subjects are students, there are fourth-year engineering students who are extensively trained as Java programmers and who have been formally taught black-box and white-box test techniques. Therefore, though replications of this experiment are needed, we believe that the results we provide in this paper and the conclusions we draw provide useful insights to practitioners and researchers alike.

6 ACKNOWLEDGMENTS

Lionel Briand's work was partly supported by a Canada Research Chair (CRC) grant. Lionel Briand and Yvan Labiche were further supported by NSERC Discovery grants. Samar Mouchawrab was supported by an Ontario Graduate Studies (OGS) graduate scholarship.

7 REFERENCES

[1] J. H. Andrews, L. C. Briand and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," *Proc. Proceedings of the 27th international conference on Software engineering*, St. Louis, MO, USA, pp. 402-411, 2005.

[2] J. H. Andrews, L. C. Briand, Y. Labiche and A. S. Namin, "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria," *Forthcoming in IEEE Transactions on Software Engineering*, 2006.

[3] E. Arisholm, L. C. Briand, S. E. Hove and Y. Labiche, "The Impact of UML Documentation on Software Maintenance: An Experimental Evaluation," *Forthcoming in IEEE Transactions on Software Engineering*, 2006.

[4] D. Baldwin and F. Sayward, "Heuristics for determining equivalence of program mutations," Yale University, Department of Computer Science 276, 1979.

[5] T. Ball, D. Hoffman, F. Ruskey, R. Webber and L. White, "State generation and automated class testing," *Software Testing, Verification and Reliability*, vol. 10 (3), pp. 149-170, 2000.

[6] V. R. Basili, F. Shull and F. Lanubile, "Building Knowledge through Families of Experiments," *IEEE Transactions on Software Engineering*, vol. 25 (4), pp. 456-473, 1999.

[7] J. M. Bieman and J. L. Schultz, "An Empirical Evaluation (and specification) of the all-du-paths testing criterion," *ACM Software Engineering Journal*, vol. 7 (1), pp. 43-51, 1992.

[8] R. V. Binder, *Testing Object-Oriented Systems - Models, Patterns, and Tools*, Addison-Wesley, 1999.

[9] K. Bogdanov and M. Holcombe, "Statechart Testing Method for Aircraft Control Systems," *Software Testing, Verification and Reliability*, vol. 11 (1), pp. 39-54, 2001.

[10] L. C. Briand, M. Di Penta and Y. Labiche, "Assessing and improving state-based class testing: a series of experiments," *Software Engineering, IEEE Transactions on*, vol. 30 (11), pp. 770-783, 2004.

[11] L. C. Briand and Y. Labiche, "A UML-Based Approach to System Testing," *Software and Systems Modeling (Springer)*, vol. 1 (1), pp. 10-42, 2002.

[12] L. C. Briand, Y. Labiche and H. Sun, "Investigating the Use of Analysis Contracts to Improve the Testability of Object-Oriented Code," *Software - Practice and Experience*, vol. 33 (7), pp. 637-672, 2003.

[13] L. C. Briand, Y. Labiche and Y. Wang, "Using Simulation to Empirically Investigate state Coverage Criteria based on Statecharts."

[14] L. C. Briand, Y. Labiche and Y. Wang, "Using Simulation to Empirically Investigate state Coverage Criteria based on Statecharts," *Proc. Proceedings of ACM International Conference on Software Engineering*, Edinburgh, Scotland, UK, pp. 86-95, May 2004, 2004.

[15] T. A. Budd and D. Angluin, "Two Notions of Correctness and their Relation to Testing," *Acta Informatica*, vol. 18 (1), pp. 31-45, 1982.

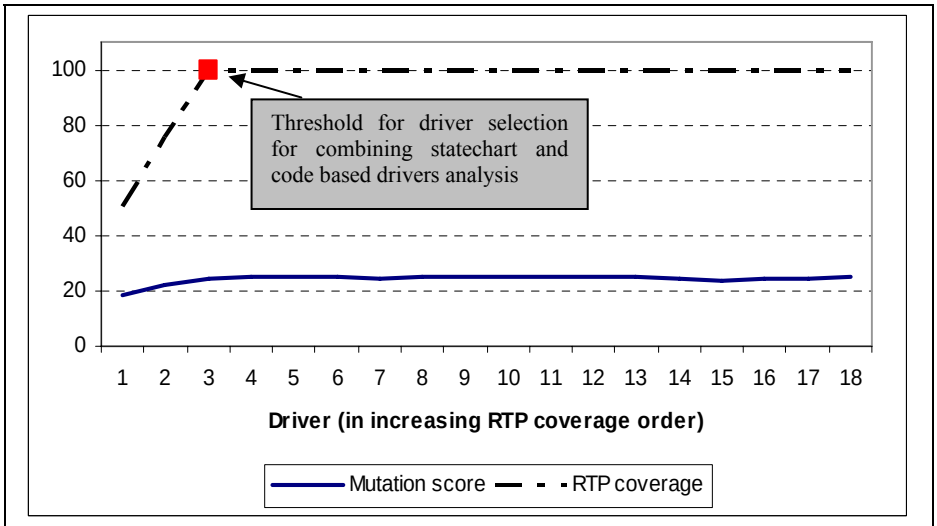
[16] P. Chevalley and P. Thevenod-Fosse, "Automated generation of statistical test cases from UML state diagrams," *Proc. Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International*, pp. 205-214, 2001.
[17] P. Chevalley and P. Thevenod-Fosse, "An empirical evaluation of statistical testing designed from UML state diagrams: the flight guidance system case study," *Proc. Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, pp. 254-263, 2001.
[18] T. S. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Transactions on Software Engineering*, vol. SE-4 (3), pp. 178-187, 1978.
[19] R. A. DeMillo, R. J. Lipton and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, vol. 11 (4), pp. 34-41, 1978.
[20] J. L. Devore and N. Farnum, *Applied Statistics for Engineers and Scientists*, Duxbury, 1999.
[21] P. G. Frankl and S. N. Weiss, "An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria," *Proc. Proceedings of the Fourth Symposium on Software Testing, Analysis and Verification*, Victoria, British Columbia, Canada, pp. 154-164, October 1991, 1991.
[22] P. G. Frankl and S. N. Weiss, "An experimental comparison of the effectiveness of branch testing and data flow testing," *Software Engineering, IEEE Transactions on*, vol. 19 (8), pp. 774-787, 1993.
[23] P. G. Frankl, S. N. Weiss and C. Hu, "All-Uses versus Mutation Testing: An Experimental Comparison of Effectiveness," *Systems and Software*, vol. 38 (3), pp. 235-253, 1997.
[24] E. Gamma, R. Helm, R. Johnson and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Computing Series, Addison-Wesley Professional, 1995.
[25] H. S. Hong, Y. G. Kim, S. D. Cha, D. H. Bae and H. Ural, "A Test Sequence Selection Method for Statecharts," *Software Testing, Verification and Reliability*, vol. 10 (4), pp. 203-227, 2000.
[26] M. Hutchins, H. Foster, T. Goradia and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," *Proc. Proceedings of the 16th international conference on Software engineering*, Sorrento, Italy, pp. 191-200, 1994.
[27] IEEE Press, "IEEE Standard Glossary of Software Engineering Technology," *ANSI/IEEE Standard 610.12-1990*, 1990.
[28] E. Kamsties and C. M. Lott, "An Empirical Evaluation of Three Defect-Detection Techniques," *Proc. Proceedings of the 5th European Software Engineering Conference*, pp. 362-383, 1995.
[29] P. S. Levy and S. Lemeshow, *Sampling of Populations: Methods and Applications*, Wiley, 3rd Edition, 1999.
[30] Y.-S. Ma, Y.-R. Kwon and J. Offutt, "Inter-Class Mutation Operators for Java," *Proc. The Thirteenth International Symposium on Software Reliability Engineering*, Annapolis, MD, November 2002, 2002.
[31] Y.-S. Ma and J. Offutt, Description of Method-level Mutation Operators for Java, http://www.isse.gmu.edu/faculty/ofut/mujava/mutopsMethod.pdf, (Last accessed

[32] Y.-S. Ma, J. Offutt and Y. R. Kwon, " MuJava : An Automated Class Mutation System," *Software Testing, Verification and Reliability*, vol. 15 (2), pp. 97-133, 2005.
[33] B. Marick, *Craft of Software Testing: Subsystems Testing Including Object-Based and Object-Oriented Testing*, Prentice-Hall, 1985.
[34] J. D. McGregor and D. A. Sykes, *A practical guide to testing object-oriented software*, Object technology series, 2001.
[35] B. Meyer, "Applying Design By Contract," *Computer*, vol. 25, pp. 40-51, 1992.
[36] C. Nebut, F. Fleurey, Y. Le Traon and J.-M. Jezequel, "Requirements by contracts allow automated system testing," *Proc. 14th International Symposium on Software Reliability Engineering, ISSRE 2003*, pp. 85- 96, 17-20 Nov. 2003, 2003.
[37] C. Nebut, F. Fleurey, Y. Le Traon and J.-M. Jezequel, "Automatic test generation: a use case driven approach," *IEEE Transactions on Software Engineering*, vol. 32 (3), pp. 140- 155, 2006.
[38] A. J. Offutt and A. Abdurazik, "Generating Tests from UML specifications," *Proc. Proc. 2nd International Conference on the Unified Modeling Language*, pp. 416-429, 1999.
[39] A. J. Offutt and W. M. Craft, "Using compiler optimization techniques to detect equivalent mutants," *The Journal of Software Testing, Verification and Reliability*, vol. 4 (3), pp. 131-154, 1994.
[40] A. J. Offutt, Y. Xiong and S. Liu, "Criteria for Generating Specification-Based Tests," *Proc. Proc. 5th International Conference on Engineering of Complex Computer Systems*, pp. 119-129, 1999.
[41] J. Offutt and A. Abdurazik, "Using UML Collaboration Diagrams for Static Checking and Test Generation," *Proc. The Third International Conference on the Unified Modeling Language (UML '00)*, York, UK, pp. 383-395, October, 2000, 2000.
[42] J. Offutt, S. Liu, A. Abdurazik and P. Ammann, "Generating Test Data From State-based Specifications," *The Journal of Software Testing, Verification and Reliability*, vol. 13 (1), pp. 25-53, 2003.
[43] J. Offutt and J. Pan, " Automatically Detecting Equivalent Mutants and Infeasible Paths," *Software Testing, Verification and Reliability*, vol. 7 (3), pp. 165-192, 1997.
[44] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating fuctional tests," *Communications of the ACM*, vol. 31 (6), pp. 676-686, 1988.
[45] P. Piwowarski, M. Ohba and J. Caruso, "Coverage Measurement Experience During Function Test," *Proc. 15th International Conference on Software Engineering, IEEE CS*, 1998.
[46] E. Weyuker, "The Cost of Data Flow Testing: An Empirical Study," *IEEE Transactions of Software Engineering*, vol. 16 (2), pp. 121-128, 1990.
[47] E. Weyuker, T. Goradia and A. Singh, "Automatically generating test data from a Boolean specification," *Software Engineering, IEEE Transactions on*, vol. 20 (5), pp. 353-363, 1994.
[48] E. J. Weyuker, "The cost of data flow testing: an empirical study," *Software Engineering, IEEE Transactions on*, vol. 16 (2), pp. 121-128, 1990.

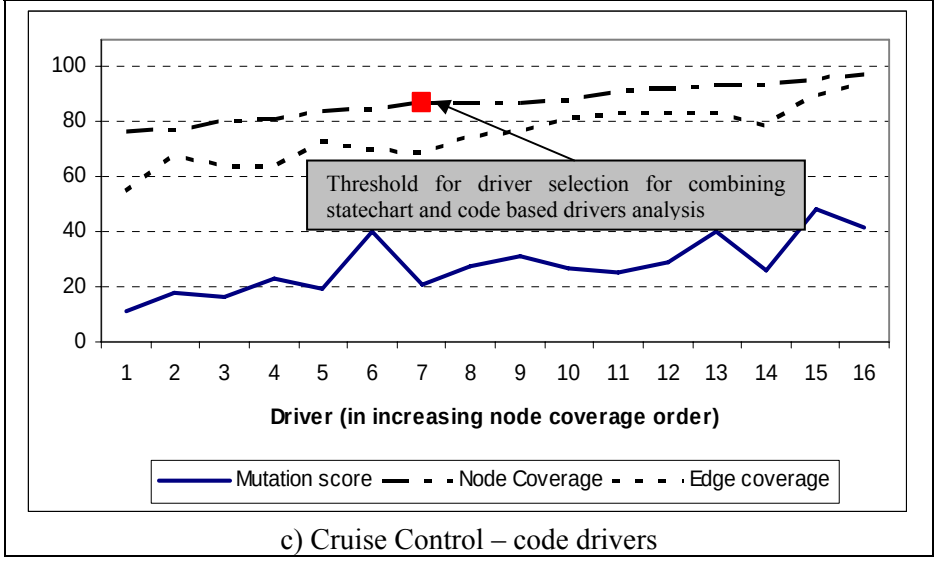
[49] C. Wohlin, P.Runeson, M. Host, M. C. Ohlsson, B. Regnell and A. Wesslen, *Experimentation in Software Engineering - An Introduction*, 2000.

[50] M. Wood, M. Roper, A. Brooks and J. Miller, "Comparing and combining software defect detection techniques: a replicated empirical study," *Proc. Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, Zurich, Switzerland, Springer-Verlag New York, Inc., pp. 262-277, 1997.

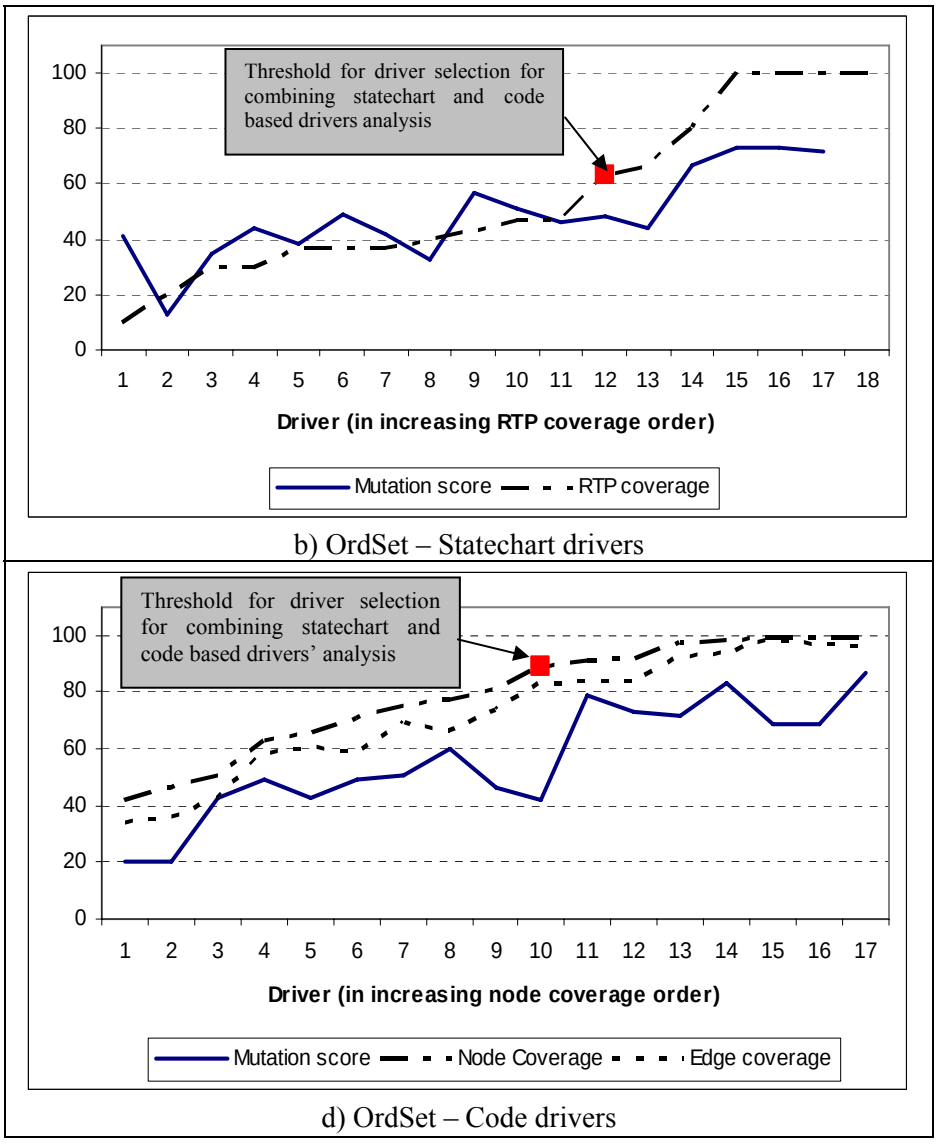
Appendix A Plot of Mutation score, Node, Edge and RTP coverage



a) Cruise Control – Statechart drivers



c) Cruise Control – code drivers



b) OrdSet – Statechart drivers

d) OrdSet – Code drivers

Figure 18: Node, Edge and RTP coverage distributions

Appendix B Descriptive statistics tables

	Cruise Control				OrdSet			
	|Fs|	**| Fc|**	**|Fs-Fc| /|F|%**	**|Fc-Fs| /|F|%**	**|Fs|**	**| Fc|**	**|Fs-Fc| /|F|%**	**|Fc-Fs| /|F|%**
Median	96	108	6.48	10.49	449	452	10.10	10.90
Mean	95	122	7.28	14.31	407	447	11.97	10.95
Min	92	80	1.55	4.40	275	264	0.16	2.08
Max	98	186	15.28	32.64	494	541	38.94	21.31
95%	98	174	15.28	32.38	483	534	34.85	19.90
90%	98	163	11.74	23.24	472	526	30.82	15.48
75%	97	146	8.81	21.50	456	499	14.46	14.42
25%	94	101	4.92	8.55	359	430	6.05	7.61
10%	93	95	3.32	7.05	292	380	0.96	4.65
5%	93	88	2.19	4.66	283	322	0.47	4.11

Table 27: Mutation scores and difference fault sets scores

	Cruise Control			OrdSet		
	|Fs∩Fc| /|F|%	**|Fs∩Fc| / |Fs|**	**|Fs∩Fc| / |Fc|**	**|Fs∩Fc| /|F|%**	**|Fs∩Fc| / |Fs|**	**|Fs∩Fc| / |Fc|**
Median	18.13	0.73	0.61	62.18	0.89	0.83
Mean	17.17	0.69	0.57	60.79	0.86	0.78
Min	8.81	0.35	0.32	35.90	0.51	0.51
Max	22.28	0.92	0.82	78.21	1.00	0.95
95%	22.28	0.90	0.81	72.83	1.00	0.93
90%	20.83	0.85	0.72	71.67	0.99	0.90
75%	19.62	0.79	0.69	66.75	0.95	0.88
25%	15.61	0.63	0.45	58.61	0.82	0.66
10%	12.88	0.52	0.34	40.13	0.66	0.58
5%	9.33	0.38	0.33	38.08	0.54	0.55

Table 28: Intersection score and ratios

	Cruise Control				OrdSet			
	|FsUFc| /|F|%	**|F – (FsUFc)| /|F|%**	**|FsUFc| / |Fs|**	**|FsUFc| / |Fc|**	**|FsUFc| /|F|%**	**|F – (FsUFc)| /|F|%**	**|FsUFc| / |Fs|**	**|FsUFc| / |Fc|**
Median	35.10	64.90	1.42	1.19	84.86	16.67	1.19	1.10
Mean	38.76	61.24	1.57	1.26	83.70	18.64	1.29	1.17
Min	28.50	43.01	1.15	1.05	72.92	11.70	1.03	1.00
Max	56.99	71.50	2.38	1.59	88.30	41.83	1.97	1.92
95%	56.74	70.98	2.29	1.59	87.84	28.13	1.80	1.78
90%	47.38	69.04	1.97	1.55	87.66	27.00	1.69	1.37
75%	46.11	67.10	1.85	1.31	86.90	21.83	1.36	1.19
25%	32.90	53.89	1.33	1.15	81.17	13.42	1.12	1.04
10%	30.96	52.62	1.26	1.10	78.38	12.42	1.08	1.01
5%	29.02	43.26	1.17	1.07	76.67	12.30	1.06	1.00

Table 29: Union and not detected faults scores and ratios

Appendix C Mutation scores per mutation operator

	Mutation scores descriptive statistics in:																	
	JDC (1)		EAM (5)		AORB (32)		AOIU (32)		AOIS (144)		ROR (79)		COI (1)		LOI (51)		ASRS (12)	
	Fs	Fs-Fc	Fs	Fs-Fc	Fs	Fs-Fc	Fs	Fs-Fc	Fs	Fs-Fc	Fs	Fs-Fc	Fs	Fs-Fc	Fs	Fs-Fc	Fs	Fs-Fc
Median	100	0	0	0	0	0	28.13	3.13	23.61	11.11	29.11	3.8	100	0	52.94	6.86	0	0
Mean	100	0	9.33	5.6	0	0	28.34	6.56	23.57	10.94	29.11	4.81	100	10	52.94	11.93	0	0
95%	100	0	20	20	0	0	28.13	0	21.53	4.17	29.11	0	100	0	52.94	0	0	0
90%	100	0	20	20	0	0	31.25	18.75	25.70	18.06	29.11	13.92	100	100	52.94	31.37	0	0
75%	100	0	20	20	0	0	29.07	18.75	25.21	18.06	29.11	13.92	100	100	52.94	31.37	0	0
25%	100	0	0	0	0	0	28.13	18.75	24.72	14.79	29.11	10.51	100	10	52.94	25.88	0	0
10%	100	0	0	0	0	0	28.13	15.63	24.31	13.02	29.11	6.33	100	0	52.94	17.65	0	0
5%	100	0	0	0	0	0	28.13	0	22.92	8.33	29.11	1.27	100	0	52.94	3.92	0	0
Min	100	0	0	0	0	0	28.13	0	22.22	6.94	29.11	0	100	0	52.94	3.53	0	0
Max	100	0	20	20	0	0	28.13	0	22.01	5.56	29.11	0	100	0	52.94	0	0	0

Table 30: Mutation scores per mutation operator statistics for Cruise Control

Only mutation operators for which a driver (code or statechart) had detected one or more faults had been included in results' tables; for example, no column has been reported for the JSI mutation operator for Cruise Control as no driver had detected any of JSI faults. For each mutation operator in the results' table, we present: (1) the number of created mutants per cluster (in brackets), (2) the percentage of detected mutants by statechart drivers of that particular mutation operator (the column named *Fs*), and (3) the percentage of detected mutants by statechart drivers, not detected by code drivers, of that particular mutation operator (the column named *Fs-Fc*). Drivers used to generate these results are those selected based on the criteria defined in Section 4.2.

	Mutation scores descriptive statistics in:																					
	JSI (5)		EAM (4)		AORB (90)		AORS (8)		AOIU (48)		AOIS (297)		AODU (4)		ROR (47)		COR (6)		COD (1)		COI (4)	
	Fs	Fs-Fc	Fs	Fs-Fc	Fs	Fs-Fc	Fs	Fs-Fc	Fs	Fs-Fc	Fs	Fs-Fc	Fs	Fs-Fc	Fs	Fs-Fc	Fs	Fs-Fc	Fs	Fs-Fc	Fs	Fs-I
Median	40	0	75	0	75.56	11.11	75	0	85.42	9.38	58.59	6.73	75	0	65.96	6.38	100	0	100	0	100	12.5
Mean	42.86	1.79	75	9.38	69.21	13.87	75	8.71	75.89	9.97	55.96	9.39	82.14	0.89	62.61	6.08	97.62	9.52	100	0	100	18.7
95%	40	0	75	0	38.89	0	50	0	50	0	38.72	0	75	0	42.55	0	83.33	0	100	0	100	0
90%	60	20	75	75	91.11	57.78	87.5	37.5	91.67	41.67	65.32	36.7	100	25	78.72	21.28	100	33.33	100	0	100	50
75%	54	20	75	75	88.78	48.06	87.5	37.5	90.42	37.5	65.12	31.4	100	0	76.81	14.89	100	33.33	100	0	100	50
25%	48	0	75	75	86.44	25.56	87.5	25	89.17	14.58	64.92	17.51	100	0	74.89	13.83	100	33.33	100	0	100	50
10%	40	0	75	0	81.67	18.89	87.5	12.5	86.46	12.5	63.64	13.22	87.5	0	71.28	10.64	100	16.67	100	0	100	31.2
5%	40	0	75	0	57.78	3.33	68.75	0	65.63	2.08	50.92	1.68	75	0	54.26	0	100	0	100	0	100	0
Min	40	0	75	0	39.56	0	57.5	0	51.25	0	41.95	0.34	75	0	46.38	0	93.33	0	100	0	100	0
Max	40	0	75	0	39.22	0	53.75	0	50.62	0	40.34	0	75	0	44.47	0	88.33	0	100	0	100	0

Table 31: Mutation scores per mutation operator statistics for OrdSet

Appendix D OrdSet statechart guard conditions

```
A:   ((v->size()).mod(min_set_size) = 0 and
     v->asSet()->size() < v->size())
   or((v->size()).mod(min_set_size)<> 0)
   or(v->size() > max_set_size and
     v->asSet()->size() < max_set_size)

B:   (s1.getSetElements()->size() +
     s2.getSetElements()->size()).mod(min_set_size) = 0 and
     s1.getSetElements()->
     intersection(s2.getSetElements())->size() <> 0)
   or(s1.getSetElements()->size() +
     s2.getSetElements()>size()).mod(min_set_size) <> 0)
   or(s1.getSetElements()t->size() +
     s2.getSetElements()->size() > max_set_size and
     s1.getSetElements()->
     union(s2.getSetElements())->size() < max_set_size)

C:   ((v->size()).mod(min_set_size) = 0 and
     v->asSet()->size() = v->size())
    or     (v->size() > max_set_size and v->asSet()->size() =
     max_set_size)

D:   (s1.getSetElements()->size() +
     s2.getSetElements()->size()).mod(min_set_size) = 0 and
     s1.getSetElements()->
     intersection(s2.getSetElements())->size() = 0)
    or     (s1.getSetElements()->
     union(s2.getSetElements())->size() = max_set_size)

E:   s1.getSetElements()->
     union(s2.getSetElements())->size() > max_set_size
```

Appendix E Contracts

E.1 Cruise Control's Contracts

E.1.1 CruiseControl class

```
context CruiseControl::handleCommand(command: String): Boolean
pre: -- none
post: result = Sequence{"engineOff", "engineOn", "accelerator",
       "brake", "on", "off","resume"}->includes(command)
```

E.1.2 SpeedControl class

```
context SpeedControl:: SpeedControl(cs: CarSpeed)
pre: not cs.ignition
post: setSpeed = 0 and state = #DISABLED

context SpeedControl::clearSpeed()
pre: -- none
post: self.speed = 0

context SpeedControl::enableControl()
pre: -- none
post: self.state = #ENABLED

context SpeedControl::disableControl()
pre: -- none
post: self.state = #DISABLED

context SpeedControl::run()
pre: -- none
post: state = #DISABLED

context SpeedControl::getState(): Integer
pre: -- none
post: result = self.state
```

E.1.3 CarSimulator class

```
context CarSimulator
inv: throttle >=0 and throttle <=maxThrottle
 and speed >= 0 and speed <= maxSpeed
 and brakepedal >= 0 and brakepedal <= maxBrake
 and ((ignition = false) implies (speed = 0 and distance = 0 and
       throttle = 0 and brakepedal = 0))

context CarSimulator::engineOn()
pre: --
post: self.ignition
```

```
context CarSimulator::engineOff()
pre: --
post: not self.ignition

context CarSimulator::accelerate()
pre: --
post: brakepedal=0 and (throttle=throttle@pre+5 or
       throttle=maxThrottle)

context CarSimulator::brake()
pre: --
post: throttle=0 and (brakepedal=brakepedal@pre+1 or
       brakepedal=maxBrake)

context CarSimulator::run()
pre: --
post: ignition = false

context CarSimulator::setThrottle(val: Integer)
pre: --
post: brakepedal=0 and
       ((val>=0 and val <= maxThrottle) implies (throttle=val))
       and (val<0 implies throttle =0)
       and (val >maxThrottle implies throttle=maxThrottle)

context CarSimulator::getSpeed(): Integer
pre: --
post: result=self.speed

context CarSimulator::getDistance(): Integer
pre: --
post: result=self.distance

context CarSimulator::getBrakepedal(): Integer
pre: --
post: result=self.brakepedal

context CarSimulator::getIgnition(): Boolean
pre: --
post: result=self.ignition

context CarSimulator::getThrottle(): Double
pre: --
post: result=self.throttle
```

E.1.4 Controller class

```
context Controller
inv: (self.controlState = INACTIVE) implies
       (self.sc.state=DISABLED and self.sc.setSpeed >= 0
        and not self.sc.cs.ignition)
     and (self.controlState = ACTIVE) implies
       (self.sc.state=DISABLED and self.sc.setSpeed = 0
        and self.sc.cs.ignition)
     and (self.controlState = CRUISING) implies
       (self.sc.state=ENABLED and self.sc.setSpeed > 0
        and self.sc.cs.ignition)
     and (self.controlState = STANDBY) implies
       (self.sc.state=DISABLED and self.sc.setSpeed > 0
        and self.sc.cs.ignition)

context Controller:: Controller(cs: CarSpeed)
pre: not cs.ignition
post: result.controlState = INACTIVE

context Controller:: brake()
pre: --
post: (controlState@pre=CRUISING) implies
       (sc.state=DISABLED and controlState=STANDBY)

context Controller:: accelerator()
pre: --
post: (controlState@pre=CRUISING) implies
       (sc.state=DISABLED and controlState=STANDBY)

context Controller:: engineOff()
pre: --
post: (controlState@pre<>INACTIVE) implies
       (sc.state=DISABLED and controlState=INACTIVE)

context Controller:: engineOn()
pre: --
post: (controlState@pre=INACTIVE) implies
       (sc.setSpeed=0 and controlState=ACTIVE)

context Controller:: on()
pre: --
post: (controlState@pre<>INACTIVE) implies
(sc.setSpeed = sc.cs.speed and sc.state=ENABLED and
controlState=CRUISING)
```

```
context Controller:: off()
pre: --
post: (controlState@pre=CRUISING) implies
       (sc.state=DISABLED and controlState=STANDBY)

context Controller:: resume()
pre: --
post: (controlState@pre=STANDBY) implies
       (sc.state=ENABLED and controlState=CRUISING)
```

E.2 OrdSet's Contracts

```
context OrdSet
inv: _set_size >= min_set_size
     and  _set_size <= max_set_size
     and  _set_size.mod(min_set_size) = 0
     and  _resize_times <= max_accepted_resizes
     and  _last < _set_size
     and  Sequence{1 .. _last+1}->
       forAll(i,j| (i<j) implies (_set->at(i) < _set->at(j))
     and  _last + 1 = self.getActualSize()
     and  _last + 1 = self.getSetElements() ->size()

context OrdSet::defSetSize(n: int): int
pre: n >= 0
post: (n < min_set_size) implies (result = min_set_size)
      and  (n >= max_set_size) implies (result = max_set_size)
      and  ((n > min_set_size and n < max_set_size) implies
        (result.mod(min_set_size) = 0  and result <
           n+min_set_size))

context OrdSet::initSetArray (v: int[])
pre: none
post: self._set_size = self.defSetSize(v->size())
      and  self._last < v->size()
      and  Sequence{1.._last+1}->forAll(i |
              v->includes(_set->at(i))

context OrdSet::resizeArray()
pre: _last = _set_size - 1
post: (_resized_times@pre < max_accepted_resizes and
       _set_size@pre + min_set_size <= max_set_size) implies
      (_set_size = _set_size@pre + min_set_size and
       _resized_times = _resized_times@pre + 1)
      and
      (_resized_times@pre == max_accepted_resizes or
```

```
        _set_size@pre + min_set_size > max_set_size) implies
        (_overflow = true)

context OrdSet::OrdSet (size: int)
pre: none
post: self._set_size = self.defSetSize(size)
      and  self._resized_times = 0
      and  _overflow = false
      and  self._last = -1
      and  self.getSetElements()->isEmpty

context OrdSet::OrdSet(v: int[])
pre: none
post: self._set_size = self.defSetSize(v->size())
      and  self._last < v->size()
      and  Sequence{1.._last+1}->forAll(i | v->includes(_set
      ->at(i))

context OrdSet::getResizedTimes(): int
pre: none
post: result = _resized_times

context OrdSet::getSetSize(): int
pre: none
post: result = _set_size

context OrdSet::getActualSize(): int
pre: none
post: result = _last + 1

context OrdSet::getSetLast(): int
pre: none
post: result = _last

context OrdSet::getSetArray(): int[]
pre: none
post: result = _set

context OrdSet::getSetElements(): int[]
pre: none
post: Sequence{0.._last}->forAll(i|result->at[i] =
       _set->at[i])

context OrdSet::isEmpty(): boolean
pre: none
```

```
post: result = _set->isEmpty()

context OrdSet::isOverflow(): boolean
pre: none
post: result = _overflow

context OrdSet::equals(x: OrdSet): int
pre: none
post:   result = Sequence{0 .. last}
       ->forAll(i|self.elementAt(i) =      x.elementAt(i)

context OrdSet::contains(n: int) : boolean
pre: none
post: result = self.getSetElements()->includes(n)

context OrdSet::contains (x: OrdSet): boolean
pre: x->notEmpty()
post: result = self.getSetElements()
       ->includesAll(x.getSetElements())

context OrdSet::remove (val: int): boolean
pre: none
post: not _overflow implies
 (result = self.getSetElements()@pre->includes(val)
 and
 not self.getSetElements()->includes(val))

context OrdSet::add(n: int)
pre: none
post: not _overflow implies
 (_set->includes(n) and (
 (!_set@pre->includes(n) and _last=_last@pre + 1) or
  (_set@pre->includes(n) and _last=_last@pre)))

context OrdSet::elementAt(where: int): int
pre: none
post: (where < 0 or where > self._last) implies (result = -1)
     and
      (where >= 0 and where <= self._last) implies (result =
       _set->at(where + 1))

context OrdSet::make_a_free_slot(n: int): int
pre: none
post: result >= 0
      and  result <= _last@pre+1)
```

```
      and  Sequence{0..result-1}->forAll(i|self.elementAt(i) =
            self@pre.elementAt(i)
      and  Sequence{result+1.._last}->forAll(i|self.elementAt(i) =
            self@pre.elementAt(i-1)

context OrdSet::union(s2: OrdSet): OrdSet
pre: none
post: result._set_size = defSetSize(self._last + s2._last + 2)
      and  not result._overflow implies result.getSetElements()->
        forAll(item | self.getSetElements()->includes(item) or
        s2.getSetElements()->includes(item))

context OrdSet::binSearch(a: int[],nElts: int, x: int): int
pre: nElts >= 0
post: (result = -1) implies (Sequence{1..nElts}->forAll(i|
       a->at(i) <> x)
       xor  a->at(result + 1) = x

context OrdSet::toString(): String
pre: none
post: none
```