Using Simulation to Empirically Investigate Test Coverage Criteria Based on Statecharts

By

Yunlan Wang, B. Eng.

**A thesis submitted to the Faculty of Graduate Studies and
Research in partial fulfillment of the requirements of the
degree of
Master of Science in ISS**

**Information and Systems Science
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, K1S 5B6
Canada**

November 2002

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-79821-6

Canada

The undersigned recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis

**Using Simulation to Empirically Investigate Test Coverage
Criteria Based on Statecharts**

Submitted by
Yunlan Wang, B. Eng.
in partial fulfillment of the requirements for
the degree of Master of Science

Chair, Professor Rafik A. Goubran

Thesis Co-Supervisor, Professor L. Briand

Thesis Co-Supervisor, Professor Y. Labiche

Carleton University

November 2002

ABSTRACT

State-based testing may be applied at any scope for object-oriented software testing. A number of testing strategies have been proposed using state machines and statecharts as test models in order to derive test sequences and validate classes or class clusters. Though such criteria have the advantage of being systematic, little is known on how cost effective they are and how they compare to each other.

This thesis presents a precise simulation and analysis procedure that involves large set of faults and the execution of large number of test sets. We investigate the cost effectiveness of four of the most referenced coverage criteria on two different representative examples, with state-dependent classes. Through the analysis of common results and differences in two case studies, more general conclusions are drawn regarding the cost benefit of using test case selection strategies based on statechart. A tool framework is developed to automate the experiment process and used to carry out our two case studies.

Acknowledgments

I would like to thank my supervisors, Dr. Lionel Briand and Dr. Yvan Labiche, for their understanding, guidance, support, and feedback.

I would like to give my sincere thanks to my parents for their support throughout my graduate studies.

I also would like to thank my friends who helped me either with suggestions or with positive affirmations.

Table of Contents

CHAPTER 1	INTRODUCTION	1
1.1	MOTIVATION	1
1.2	CONTRIBUTIONS OF THIS THESIS	2
1.3	THESIS CONTENTS	3
CHAPTER 2	FUNDAMENTAL CONCEPTS AND DEFINITIONS	4
2.1	BASIC DEFINITIONS	4
2.2	STATECHARTS AND TEST COVERAGE STRATEGIES	5
CHAPTER 3	EXPERIMENTAL DESIGN	9
3.1	MEASUREMENT	10
3.2	SIMULATION PROCESS	11
3.3	DATA ANALYSIS	18
3.4	CASE STUDIES	20
3.4.1	<i>Case study 1: Cruise Control</i>	<i>21</i>
3.4.2	<i>Case study 2: OrdSet</i>	<i>23</i>
3.5	MUTATION OPERATORS	27
3.5.1	<i>Case Study 1: Cruise Control</i>	<i>28</i>
3.5.2	<i>Case Study 2: OrdSet</i>	<i>29</i>
CHAPTER 4	RESULTS	31
4.1	CASE STUDY 1: CRUISE CONTROL	32
4.1.1	<i>Cost-Benefit Analysis (Cruise Control)</i>	<i>32</i>
4.1.2	<i>Further Comparison of the Three Coverage Criteria (Cruise Control)</i>	<i>35</i>
4.1.3	<i>Comparing the mutation score distributions of the three Criteria (Cruise Control)</i>	<i>36</i>
4.1.4	<i>Further Comparing Mutation Score Distributions (Cruise Control)</i>	<i>38</i>
4.1.5	<i>Detailed Analysis of Live Mutants (Cruise Control)</i>	<i>41</i>
4.1.6	<i>New Termination Criterion for Transition Trees</i>	<i>44</i>
4.2	CASE STUDY 2: ORDSET	47
4.2.1	<i>Cost-Benefit Analysis (OrdSet)</i>	<i>47</i>
4.2.2	<i>Further Comparison of the Four Coverage Criteria (OrdSet)</i>	<i>50</i>
4.2.3	<i>Comparing the mutation score distributions of the four Criteria (OrdSet)</i>	<i>51</i>
4.2.4	<i>Further Comparing Mutation Score Distributions (OrdSet)</i>	<i>53</i>
4.2.5	<i>Detailed Analysis of Live Mutants (OrdSet)</i>	<i>55</i>
4.3	CONCLUSIONS FROM THE TWO CASE STUDIES	56
CHAPTER 5	THREATS TO VALIDITY	58
CHAPTER 6	SIMULATION TOOL FRAMEWORK	59
6.1	REQUIREMENTS FOR IFTR	59

6.2	SYSTEM ANALYSIS AND DESIGN FOR IFTR.....	62
6.2.1	<i>Use case diagram</i>	63
6.2.2	<i>Subsystem decomposition</i>	65
6.3	IMPLEMENTATION	78
6.4	USAGE.....	78
6.5	CARRYING OUT THE EXPERIMENT WHEN THE STATECHART HAS GUARD CONDITIONS AND PARAMETERS	83
6.6	EXTENDING IFTR.....	86
6.7	SUMMARY	88
CHAPTER 7	CONCLUSIONS	90
APPENDIX A	CLASS DIAGRAM FOR CRUISE CONTROL SYSTEM.....	96
APPENDIX B	CLASS DIAGRAM FOR ORDSET	97
APPENDIX C	COVERAGE RATIO VERSUS CUMULATIVE LENGTH (CRUISE CONTROL).....	98
APPENDIX D	COVERAGE RATIO VERSUS CUMULATIVE LENGTH (ORDSET).....	99
APPENDIX E	MUTANTS DESIGNED FOR CRUISE CONTROL CASE STUDY	100
APPENDIX F	MUTANTS DESIGNED FOR ORDSET CASE STUDY.....	104
APPENDIX G	ANALYSIS OF LIVE MUTANTS WITH TRANSITION TREES (CRUISE CONTROL).....	107
APPENDIX H	ANALYSIS OF LIVE MUTANTS WITH ATP, TT AND FP ADEQUATE TEST SETS (ORDSET).....	112
APPENDIX I	SYSTEM INPUTS AND OUTPUTS FOR IFTR	116
APPENDIX J	USE CASE DESCRIPTION FOR IFTR.....	119

List of Figures

Figure 1. Step 1: Preparation of Simulation.....	13
Figure 2. Step 2: Test Simulation.....	17
Figure 3. Statechart for Cruise Control	22
Figure 4. Transition Trees (Cruise Control).....	23
Figure 5. Statechart for OrdSet	25
Figure 6. Transition Tree (OrdSet).....	27
Figure 7. Fault Distributions across Mutation Operators (Cruise Control).....	29
Figure 8. Fault Distributions across Mutation Operators (OrdSet).....	30
Figure 9. Mutation Score versus Cumulative Length (Cruise Control)	34
Figure 10. Bivariate Relationships (Cruise Control).....	35
Figure 11. Mutation Score Distributions for Adequate Test Sets (Cruise Control).....	37
Figure 12. Distributions of Mutation Scores when Coverage is > 90% (Cruise Control)	40
Figure 13. Modified Transition Tree (Cruise Control)	45
Figure 14. Bivariate Relationships with all criteria (Cruise Control)	46
Figure 15. Distributions of Mutation Scores when Coverage is > 90% with TTM (Cruise Control).....	47
Figure 16. Mutation Score versus Cumulative Length (OrdSet)	49
Figure 17. Bivariate Relationships (OrdSet).....	51
Figure 18. Mutation Score Distributions for Adequate Test Sets (OrdSet)	52
Figure 19. Distributions of Mutation Score when Coverage is > 90% (OrdSet)	54
Figure 20. Overview of IFTR.....	61
Figure 21. Use case diagram for IFTR.....	63
Figure 22. Subsystem decomposition for IFTR	66
Figure 23. IFTRGUI subsystem class diagram	68
Figure 24. ModelTransformation subsystem class diagram	69
Figure 25. MutantGenerationElimination subsystem class diagram.....	72
Figure 26. TestCaseGenerationExecution subsystem class diagram	74
Figure 27. TestSimulation subsystem class diagram.....	76
Figure 28. System Menu GUI of IFTR.....	79
Figure 29. Model Transformation submenu.....	80
Figure 30. Select Criterion dialog box	81
Figure 31. Mutant Generation submenu.....	81
Figure 32. Test Case Generation Execution submenu	82
Figure 33. Test Simulation submenu.....	83
Figure 34. The equivalent statechart for OrdSet	85
Figure 35. Class diagram for Cruise Control System.....	96
Figure 36. Class diagram for OrdSet.....	97
Figure 37. Coverage Ratio versus Cumulative Length (Cruise Control)	98
Figure 38. Coverage Ratio versus Cumulative Length (OrdSet)	99

List of Tables

Table 1. Mutation Score Distribution Statistics for all Adequate Test Sets (Cruise Control)	38
Table 2. Mutation Score Distribution Statistics for Test Sets with Coverage > 90% (Cruise Control).....	41
Table 3. Live mutants for transitions trees (Cruise Control).....	42
Table 4. Mutation Score Distribution Statistics for TTM Test Sets with coverage > 90% (Cruise Control).....	47
Table 5. Mutation Score Distribution Statistics for all Adequate Test Sets (OrdSet).....	52
Table 6. Mutation Score Distribution Statistics for Test Sets with Coverage > 90% (OrdSet).....	53
Table 7. Live Mutants with ATP, TT and FP Adequate Test Sets (OrdSet).....	55
Table 8. Mutant list for Cruise Control case study	103
Table 9. Mutant list for OrdSet case study.....	106
Table 10. Live Mutants with Transition Trees (Cruise Control).....	107
Table 11. Live Mutants with ATP, TT and FP Adequate Test Sets (OrdSet).....	112

Chapter 1 Introduction

1.1 Motivation

A number of papers [Chow 1978, Binder 1999, Offutt 1999a] have proposed using state machines and statecharts as test models in order to derive test sequences and validate classes or class clusters. Offutt performed an experiment where simple statecharts coverage criteria were evaluated [Offutt 1999a]. Binder proposed another method adapted from Chow [Chow 1978] where statecharts are traversed to derive transition trees [Binder 1999] and each path in the transition tree is to be tested. One issue is that, though such criteria have the advantage of being systematic, little is known on how cost effective they are and how they compare to each other. Since faults are much less expensive to be corrected when caught during class testing, it is important to ensure that effective strategies are being used to detect them, as long as the cost is deemed acceptable.

Determining the effectiveness of these criteria cannot be performed by analytical means [Weyuker 1991]. Like other types of criteria (e.g., data flow criteria [Frankl 1993]), experimental evaluation is required. But performing such experiments presents a number of challenges. First, sets of test cases (test sets) fulfilling a given coverage criterion are not consistent in the sense that they may result in different fault detection effectiveness. There is a certain degree of randomness to such criteria and random variations should be accounted for when comparing criteria. Second, evaluating how effective coverage criteria are at uncovering faults requires the presence of large numbers

of faults in the classes under test, if we are to draw probabilistic conclusions and compare criteria using statistical inference testing. There are usually not enough faults present in real software systems as it would translate into unrealistic fault densities.

In order to experimentally investigate the cost effectiveness of statechart coverage criteria, we resort to fault seeding and simulation so as to address the abovementioned issues. We propose a precise simulation and analysis procedure, based in part on the literature reporting test experiments, and investigate the cost effectiveness of four coverage criteria on two class clusters exhibiting a class-dependent behavior.

1.2 Contributions of This Thesis

The contributions of this thesis consist of three aspects.

First, this thesis proposes a precise simulation and analysis procedure for the empirical study, based in part on the literature reporting test experiments. A large set of faults and a large number of test sets are involved to perform the simulation. The data resulting from the simulation is analyzed in a number of different and complementary ways.

Second, this thesis investigates the cost effectiveness of four of the most referenced coverage criteria based on UML statechart. Two case studies were selected to compare the coverage criteria in terms of cost and fault detection effectiveness. These two examples represent two typical situations encountered in practice: (1) control classes in reactive, real-time systems, (2) classes encapsulating complex data structures with access functions. The overall conclusions are drawn by identifying the common results and explaining the differences.

Third, a test simulation tool framework has been developed and implemented to automate the experiment process. Both case studies were performed by using this tool. Although the current version of the tool made several assumptions regarding the characteristics of statechart, it can be extended without major impact on the existing implementation.

1.3 Thesis Contents

Chapter 2 introduces the fundamental concepts and definitions used throughout this thesis. Chapter 3 presents our experimental methodology and the specifics of our case studies. Chapter 4 reports on the results of our case studies regarding the cost-effectiveness of the statechart-based strategies being investigated. Chapter 5 discusses the threats to validity of our study. Chapter 6 describes the main features and design aspects of the simulation tool, and Chapter 7 concludes by summarizing our findings and outlining future work.

Chapter 2 Fundamental Concepts and Definitions

We present below the basic concepts and definitions used throughout this thesis.

2.1 Basic Definitions

A number of basic concepts and definitions are used throughout this thesis and in the testing literature [Weyuker 1991, Frankl 1993, Hutchins 1994]. Most test strategies are model-based, i.e., a model of the Software Under Test (SUT) is developed and coverage measures based on that model are defined. Such a coverage measure is also referred to as an *adequacy criterion*. The test strategy then consists in reaching 100% coverage with a test set of minimal size. In our context, the test model is derived based on a statechart and coverage is defined in terms of transitions, transition pairs, or specific transition sequences to be covered.

The *coverage ratio* indicates the proportion of test requirements of a given adequacy criterion that is satisfied by a test set. For instance, if we wish to cover all the transitions of a statechart (All Transitions criterion), the coverage ratio is the proportion of the transitions covered by the test set. There may exist many different test sets that yield the same coverage ratio. A test set is considered *C-adequate* when the coverage ratio achieves 100% for a given criterion *C*.

It is very common when experimenting with test strategies and adequacy criteria that faults be seeded in the SUT [Mathur 1994, Frankl 1997]. The main motivation is to have a large enough fault population so as to be able to compare adequacy criteria and determine their differences in effectiveness. Each program where a fault is seeded is

referred to as a *Mutant*. Usually, in order to avoid interaction effects between faults, one and only one fault is seeded per mutant program. In order to be systematic and as complete as possible, faults are usually seeded by using mutation operators, based on a complete classification of faults for a given programming language [King 1991, Kim 1999, Kim 2000]. When a test case yields a failure on a mutant program, the test case is said to *kill* the mutant. It is usual to talk about *killed* and *live* mutants. The *mutation score* is defined as the number of mutants killed by a given test set divided by the total number of mutants that were generated. This score is a typical measure of *effectiveness* when assessing adequacy criteria. Note that it is common that some mutants be functionally equivalent to the original program. So when a mutant is not killed by a test set, one needs to check whether it is *equivalent*. Equivalent mutants should not be counted when calculating the mutation score of a given test set.

A usual surrogate measure of cost is the *size* of test sets. The underlying assumption is that test cost is proportional to the test set size. There is not one way of measuring size and we discuss in further details in Section 3.1 the choices we have made.

For a complete assessment of any adequacy criterion, results usually must be compared with what would be obtained by chance [Frankl 1993], that is when selecting test cases at random. This is usually referred to as the *null criterion*.

2.2 Statecharts and Test Coverage Strategies

In order to define adequacy criteria on statecharts, it is necessary to remove all hierarchy and concurrency in them. Binder refers to this process as *flattening* statecharts [Binder 1999]. For statecharts characterizing class (or class cluster) behavior, this should be feasible without leading to unmanageable complexity. A number of criteria have then

been defined to “cover” statecharts [Binder 1999, Offutt 1999a, Hong 2000]. The simplest one simply requires covering all the transitions in the statechart. However, to uncover certain faults, it is necessary to execute specific *sequences* of transitions [Kung 1994].

We use here the definitions given by Offutt [Offutt 1999b]. The *All Transitions* criterion requires the test set to cover every transition in the statechart. For *All Transition Pairs*, the test set must contain every pair of adjacent transitions. When there are guard conditions on transitions, the *Full Predicate* criterion can also be applied. Adequate test sets must then include test cases that cause each clause¹ in every guard condition to control the value of the guard and to drive it to true. The rationale is that each clause should be tested independently, without the overall guard condition being influenced by other clauses. This approach ensures that any fault in the implementation of any clause will be detected and not masked by other clauses, e.g., a clause implementation is faulty and evaluates to false when it should be true, but the overall guard still evaluates to true.

But from a practical standpoint, it may not be possible to cover all possible sequences of transitions especially that this set may be infinite when there are cycles of transitions. So the question then becomes: What should be the sequences of transitions to be covered? For example, Offutt [Offutt 1999a] experimented with all transition pairs. But sequences of any length could be considered, though this usually leads to an unmanageable number of sequences as in many cases even testing all transition pairs leads to a substantial effort.

¹ A clause is a Boolean expression that does not contain Boolean operators.

Chow [Chow 1978] proposed the W-method that was originally defined for regular state machines. Binder [Binder 1999] then adapted it to UML statecharts. This technique consists in traversing the graph corresponding to the statechart (where nodes and edges are states and transitions, respectively) following a precise procedure and building the tree corresponding to this traversing, referred to as a *transition tree*. The adequacy criterion then becomes the full coverage of the tree paths. More precisely, the traversing is performed using a breadth-first search algorithm and a tree node is considered terminal when the state it represents is already presented anywhere in the tree (i.e., at the same level or above in the transition tree). Binder defines the set of paths covered by the tree as *round-trip paths* as they capture all transition sequences that begin and end with the same state (with no repetitions of states other than the sequence start and end state) and *simple paths* from the initial to the final state of the statechart. When there are guard conditions on transitions, the traversing depends on the structure of the predicates forming the guards. If a guard is a simple predicate or only contains AND operators, only one test is needed to test the guard and ensure that each clause correctly evaluates to true. If the predicate contains disjunctions (OR operators), one test is required for each truth value combination (of clauses) that makes the guard true. The round trip path test suite is augmented by the test of sneak paths, i.e., unspecified transitions and situations in which the predicate of a guarded transition evaluates to false, in the case the statechart is not completely specified (i.e., not all event/state pairs are explicitly shown in the statechart) [Binder 1999]. Testing sneak paths then first consists in placing the system under test in each of the state: e.g., using a sequence of events to reach the desired state, or using built-in test support (i.e., a mechanism to directly set the state of the system

under test). Then, for each state, all the illegal events are applied and one verifies that the state is unchanged.

Note that the W-method consists of two steps: the first one traverses the transition tree, and the second one appends a state identification sequence to each transition tree sequence in order to check the state that was reached. Other similar techniques for state based testing from finite state machines exist (e.g., [Lee 1996]), but they all test the same sequences and only differ with respect to the sequence added for the state identification problem. In the context of class testing, the adaptation performed by Binder reuses the first part and assumes it is possible to directly check the state invariant, e.g., defining/using a particular method that verifies it, and thus replace the identification sequence with a call to a state invariant method. The implementation must have a trusted ability to report the resultant states [Binder 1999].

According to Binder, this technique will find all missing transitions, outputs, and actions, and will detect some of the corrupt states. But it is a priori difficult to know how it compares to criteria such as all transitions or all transition pairs in terms of cost and fault detection.

Chapter 3 Experimental Design

Our main objective is to assess and compare the cost-effectiveness of four adequacy criteria based on statecharts: all transitions, all transition pairs, all paths in transition trees and full predicate. Like other types of test coverage criteria, the comparison of these state-based criteria cannot be done by analytical means and experimental evaluation is required. In Section 3.1, we discuss the measurements used to evaluate the cost-effectiveness of test coverage criteria and Section 3.2 describes the experimental process in a precise manner. A tool framework has been developed to support and automate this process and its functionality, architecture, and design are presented in Chapter 6.

We investigate the typical cost incurred to achieve them and since this is rather specific to the example system under study, we focus on the *relative* cost of these adequacy criteria. We also compare their capability to detect faults and then want to determine what criterion, among the four under study, can be recommended based on the quantitative results we obtain (see Sections 3.3). While our results are by definition specific to our case studies, we experiment with two different typical state-based behavior classes – the control classes in a reactive system, and a container class (Section 3.4) – and we hope that results can be generalized to such a context. As discussed in Section 3.5, we seed a large variety of faults in order to help improve the generality of results with respect to fault types.

Besides quantitative analysis, we also investigate why certain faults are not detected and what test cases would have been able to detect them (see Chapter 4). This can provide us with insight on how certain adequacy criteria can be improved.

3.1 Measurement

We are looking at the cost-effectiveness of adequacy criteria based on statecharts. As mentioned above, effectiveness will be measured in terms of mutant programs being killed. Those mutants will be seeded using the operators described in Section 3.5, where further justification is provided. Equivalent mutants are removed from the computation of effectiveness. Though various mutants can lead to failures of different severity, it is difficult to account for such factor in the context of an experiment such as the one presented here. Severity is fundamentally a context-dependent, subjective notion, and is out of the scope of this thesis.

Measuring the cost of testing in such an experimental context is also a challenge. In many papers, the size of test sets [Hutchins 1994] has been usually adopted as a surrogate measure, making the assumption that cost is overall proportional to test set size. The notion of cost in the context of testing is a complex one as one may want to consider time to market, computer time usage as part of the equation [Weyuker 1991]. In our context, test set size can be simply measured by counting the number of test sequences in a test set. However, not all test sequences contain the same number of method invocations. We could go further and say that not all sequences take the same time to execute. But without going to that level of detail, we could measure size as the number of method invocations in the test set. We refer to this as the *cumulative length* of a test set since it is the result of summing up the number of method invocations in each test

sequence. Note, however, that all the results presented in this thesis are very similar when simply counting the number of test sequences.

Another important measure here is coverage. We investigate the following adequacy criteria: All transitions (AT), All transition pairs (ATP), and all the paths in the transition tree (TT). When relevant, we also investigate Full Predicate (FP) coverage. For AT, ATP and FP we simply measure coverage as the percentage of transitions and transition pairs, respectively, that are executed by the test set. As for TT, we measure the percentage of tree paths that are covered by the test set.

3.2 Simulation Process

Assessing the cost-effectiveness of an adequacy criterion C requires to be able, for a given system and set of faults, to draw conclusions regarding the probability of detecting a fault of a C -adequate test set. In order to do that, it is necessary to work with a large set of faults, or at least with fault densities that are usually higher than in commercial software. It is difficult, for example, to draw any conclusion regarding the test of a class if we are working with 10 faults. A 10% effectiveness difference between two criteria would result, on average, into a one-fault difference. Moreover, this difference might only be observable if we run a number of different test sets. However, a 10% effectiveness improvement, if the cost of the two criteria is equivalent, may constitute a practically significant improvement.

Another issue is related to the fact that C -adequate test sets may not be consistent in terms of effectiveness. This means that the effectiveness, for a given criterion C , will be characterized by a probability *distribution* of mutation scores. To empirically determine such a distribution, we need to run a large number of test sets, all of them

satisfying the criterion under study. Those test sets may also differ in terms of size, e.g., cumulative length, and therefore lead to a size probability distribution.

From these observations, it becomes clear that to study the cost-effectiveness of adequacy criteria, including statechart-based criteria, we need to perform simulations involving large numbers of faults (i.e., mutant programs) and the execution of large numbers of adequate test sets. We describe below the simulation procedure we followed, which is supported by a flexible simulation tool framework that is developed by the author. This procedure is based in part on the experimental testing literature [Weyuker 1993, Hutchins 1994, Mathur 1994] and is adapted to our objectives and needs.

We split the procedure into two large steps: (1) Preparation, (2) Simulation. The substeps in Preparation can be described as follows and are illustrated in Figure 1:

- P-1 Specify the criteria to be investigated, the SUT and the UML statechart. In our experiments, we investigate three criteria (AT, ATP, TT), plus FP when relevant.
- P-2 Analyze the test requirements, i.e., the features that need to be covered for each adequacy criterion. As control flow and data flow coverage criteria [Weyuker 1993], the test pool may not be able to reach 100% coverage for state-based test criteria since guard conditions in statecharts may render certain transition sequences impossible [Binder 1999]. We therefore define *C*-adequate test sets as the test sets that cover all *feasible* test requirements. This is not an issue in the statechart of our first case study (Cruise control, Section 3.4.1) where all transition pairs and transition tree paths can be covered. But the second case study (OrdSet) shows such examples of infeasible paths (Section 3.4.2).

Figure 1. Step 1: Preparation of Simulation

- P-3 Generate mutant programs using a well-defined set of mutation operators. For a given SUT, all *applicable* mutation operators for the given program must be covered. The distribution of mutants across operators is usually a function of the program characteristics (e.g., usage of polymorphism) and should ideally reflect distributions observed during program development.
- P-4 Create the pool of test cases (Test Pool – TP) to be used during simulation. The test cases in the initial test pool are generated automatically by randomly traversing the statechart so as to obtain a large number of possible transition

sequences. Every test case in the test pool is a transition sequence that starts from the initial state. Additional test cases may be added if the test pool is not adequate for all selected criteria (see Step P-7). It is important that a 100% coverage can be reached for each criterion with the test pool. Moreover, the size of the test pool has to be sufficient so as to generate a large variety of adequate test sets for each adequacy criterion. But an extremely large test pool would make running the simulation impractical as all test cases have to be executed on all mutant programs. In our context, cycles of transitions in statecharts allow for arbitrarily long test sequences. Therefore, a maximum length needs to be a priori imposed on generated test sequences. This is driven by two main factors. On one hand, there must be long enough sequences so that all transitions (pairs) can be reached and that all transition tree paths can be covered. On the other hand, it is obvious that very long sequences will be more likely to kill a large number of mutants. In situations where one test case kills a large proportion of mutants we have encountered difficulties observing relationships between test set size, coverage ratio, and mutation score. For our two case studies, accounting for those factors, we set the maximum length of test sequences to 7 and 11, respectively.

- P-5 Build the *coverage matrices* and determine the coverage of the TP. In the coverage matrix, each row represents a test case in the test pool and each column represents a testing feature (e.g., a transition sequence that must be covered). If test case i covers feature j , an entry of “1” is put in position (i, j) . In our context, we determine which transitions, transition pairs, and tree path are covered by each test case.

- P-6 Check whether the initial test pool is adequate for every criterion chosen in the experiment. If the TP is adequate, then enter Step P-8.
- P-7 If the TP does not satisfy any one of the adequacy criteria selected in the experiment, then additional test cases need to be defined so as to cover those features (e.g., transition pairs) that are not covered by the test pool in the coverage matrix. A tool is required to achieve complete coverage by automatically generating test cases that execute uncovered features, e.g., transitions, pairs. Automation is further discussed in Section 6.2.2.4.
- P-8 Prune the test pool by eliminating identical test sequences (which can be the result of mistakes while manually adding test cases) to get a smaller test pool with the same coverage of 100%. For our two case studies, test pools are composed of 1339 and 14864 test cases, respectively. The pool of the second case study is much larger as some events have parameters and there are guard conditions on some of the transitions (Section 3.4.2).
- P-9 Run all test cases in the test pool on the original version of the program to create the test oracle. The expected responses of each trigger event in the test sequences are recorded in a file.
- P-10 Execute all test cases in the test pool on the mutant programs. The actual outputs are compared with the test oracle. If the actual outputs differ from the expected outputs for a given test case, the corresponding mutant is considered to be killed by this test case.
- P-11 Build the *result matrix* which indicates which mutants are killed in Step P-10. Each row in the result matrix also corresponds to a test case in the test pool, and

each column corresponds to a mutant. An entry of “1” in position (i, j) indicates that mutant j is killed by test case i . The order of rows in the coverage matrices and the result matrix are identical.

P-12 If a mutant is not killed by any test case, one must check whether it is an equivalent mutant and if so, make sure it is not accounted when computing mutation scores.

P-13 Enter the second phase, that is the simulation proper, which will make use of the coverage matrices and the result matrix to generate adequate test sets and compute their corresponding size and effectiveness.

The actual simulation phase is generating 100 adequate test sets, for a given adequacy criterion, so as to be able to analyze frequency distributions, perform statistical testing and perform comparison across adequacy criteria (see Section 3.3). As shown in Figure 2, for each test set, we iteratively select a new test case, which is not yet part of the test set, and that increases the ratio the most: increase in coverage / increase in cumulative length. In other words, we use a heuristic to derive the most cost-effective test sets. If several candidates show the same ratios, one test case is randomly selected. This general procedure simulates what a statechart based test tool would do to select a minimum number of test cases to fulfill a criterion. In other words, we attempt to converge as fast as possible towards full coverage. A test set is complete when it is adequate for the selected criterion.

The steps in Figure 2 can be described as follows:

S-1 Initialize a new empty test set T .

- S-2 Rank the test cases according to the increase of coverage they yield if added to T divided by the length of the test case. This is referred to as the *normalized* increase in coverage. Some test cases may have a tied rank.
- S-3 Randomly select a test case from those that have the highest normalized increase in coverage and add it to T .

Figure 2. Step 2: Test Simulation

- S-4 Determine the coverage ratio of the current test set T by counting the test features it covers in the coverage matrix corresponding to the chosen test criterion and record this coverage ratio.
- S-5 Determine the mutation score of T by first counting the number of mutants killed by the current test set T in the result matrix, and then dividing this number by the total number of non-equivalent mutants. Record the mutation score.

- S-6 Check if the test set T is adequate. If T covers all the features in the coverage matrix, enter Step S-7. If T is not adequate, return to Step S-2.
- S-7 Record the test set size, cumulative length, coverage ratio, and mutation score after adding each test case into T .
- S-8 Check if the desired number of C -adequate test sets is obtained. Repeat from Step S-1 to Step S-7 until this number is reached, e.g., 100 in our case studies.

3.3 Data Analysis

In order to achieve our research objectives, we analyze the data resulting from the simulation in a number of different and complementary ways. First, we look at the relationship between cost (test set cumulative length) and benefit (mutation score), for each of the four adequacy criteria (Sections 4.1.1 and 4.2.1 for the two case studies respectively). The objective is to first graphically determine the percentage of mutants that can be expected to be killed for a given coverage ratio and for adequate test sets. In order to graphically display the trend, we plot for each criterion the data points corresponding to the 100 simulation runs. We then fit smoothing splines [Eubank 1988] to identify an optimal fit curve. The cubic spline method uses a set of third degree polynomials spliced together such that the resulting curve is continuous and smooth. This is an excellent way to get an idea of the shape of the relationship between two variables (e.g., mutation score and cumulative length) without making assumptions regarding the functional form of that relationship. We also compare the splines we obtain and the uncertainty around them with what we would obtain by randomly selecting test cases

until we reach an equivalent or superior² cumulative length. This is referred to as the *Null criterion*, as opposed to adequacy criteria. This comparison allows us to get a first insight into whether coverage-driven test case selection can bring benefits over random test case selection, assuming a test pool was generated from the statechart.

To better look at the relative cost and benefit of the four adequacy criteria, we plot all their respective splines on one graph (Sections 4.1.2 and 4.2.2, respectively). We do that for (1) coverage vs. cumulative length, (2) mutation score vs. coverage ratio, (3) mutation score vs. cumulative length. Analyzing (1) tells us about the cost of reaching perfect coverage whereas (2) tells us about the level of fault detection we can hope to achieve for a given coverage ratio. Subsequently, (3) tells us about the cost-effectiveness of different adequacy criteria at various coverage ratio levels.

The next step is to compare the distributions of mutation scores across the sets of adequate test sets of each adequacy criterion (Sections 4.1.3 and 4.2.3, respectively). We want to know whether adequate test sets for the four considered criteria yield significantly different results in terms of mutation scores, i.e., the risk of not detecting a fault. We therefore statistically compare the distributions obtained based on the 100 simulation runs. We determine whether the means are significantly different using a t-test³ [JMP 2000].

Another way to analyze mutation score distributions for adequate test sets is to compare them to what is obtained with the random selection of test cases (Null criterion) when forming test sets of equivalent cumulative length (Sections 4.1.4 and 4.2.4, respectively). For each of the adequacy criteria, we select all test sets (among the 100

² With random selection, we cannot always reach *exactly* that cumulative length.

adequate test sets) that are above the 90% coverage ratio threshold. We determine the minimum cumulative length associated with them. We then select all test sets in Figure 9 (respectively Figure 16) that are above this minimum size, thus selecting a subset of the Null criterion observations that are at least as large as the smallest test set that reaches 90% coverage. The mutation score distributions of those two sets of observations are then compared. The rationale is to see whether, for high coverage values, coverage-driven selection of test cases perform better (i.e., show a lower risk of not detecting a fault) than random selection, when the cumulative lengths of test sets are comparable. Our comparison procedure tends to be conservative (i.e., advantage the Null criterion) as the subsets of Null criterion test sets that were selected tend to show higher, average cumulative lengths⁴. As discussed above, the reason is that when generating the Null criterion test sets, we randomly select test cases until we reach or go over the *maximum* cumulative length of the corresponding adequacy criterion test sets.

3.4 Case Studies

Our two case studies have been selected because they are representative of two typical situations where statecharts are used. The first one is a class cluster in a real-time system (Cruise Control) whereas the second one is implementing a data structure (Ordered sets). Both situations require the modeling of state behavior, for example using UML statecharts. By using very different examples, we hope through the analysis of common results and differences, to derive more general conclusions.

³ The t-test becomes conservative when distributions depart from normality. However, in our case, since results turned out to be significant, there was no need to consider other, non-parametric statistical tests.

⁴ 23.5, 93.4, 63.7 vs. 27, 102.9, 67.5 for AT, ATP, and TT, respectively for the first case study; and 24.1, 68.2, 36, 79.5 vs. 27.3, 73.4, 39.5, 84.8 for AT, ATP, TT and FP, respectively, for the second case study.

3.4.1 Case study 1: Cruise Control

We use a small Cruise Control System [Magee 1999] which is implemented in Java and contains six classes, namely `CruiseControl`, `CruiseDisplay`, `CarSpeed`, `Controller`, `SpeedControl` and `CarSimulator`. The class diagram is shown in Appendix A. Class `CruiseControl` and `CruiseDisplay` are related to the Graphic User Interface (GUI) to accept mouse clicks and display the results⁵. Class `CarSpeed` is an abstract class. The other three are the core classes of the system that implement the control functions and are the classes we will focus our attention on. There is no inheritance among these three classes. The statechart representing their state-dependent behavior is represented in Figure 3. Note that in this statechart, transitions do not have guard conditions, and events do not have parameters.

By following the algorithm presented in [Binder 1999], three possible transition trees are generated as shown in Figure 4. The traversal algorithm of the statechart described by Binder (Section 2.2 above) to build the tree is not deterministic: the result depends on the way one traverses the graph corresponding to the statechart during the breadth first search. However, these transition trees are all equivalent in the sense that they allow the coverage of all the round-trip paths in the statechart.

⁵ As we focus here on functional testing, the GUI in the original implementation of the cruise control system is removed when performing our experiment. Our test drivers directly interact with the non-GUI part of the system.

Figure 3. Statechart for Cruise Control

Figure 1: Transition Tree 1 (TT1):

```

graph LR
    Idle1[Idle] -- engineOn --> Running1[Running]
    Running1 -- accelerator --> Running2[Running]
    Running1 -- brake --> Running3[Running]
    Running1 -- on --> Cruising1[Cruising]
    Running1 -- engineOff --> Idle2[Idle]
    Cruising1 -- accelerator --> Standby1[Standby]
    Cruising1 -- brake --> Cruising2[Cruising]
    Cruising1 -- on --> Cruising3[Cruising]
    Cruising1 -- engineOff --> Standby2[Standby]
    Cruising1 -- off --> Standby3[Standby]
    Standby1 -- accelerator --> Cruising4[Cruising]
    Standby1 -- resume --> Cruising5[Cruising]
    Standby1 -- brake --> Standby4[Standby]
    Standby1 -- engineOff --> Standby5[Standby]
    Standby1 -- off --> Idle3[Idle]
  
```

The diagram illustrates a state transition tree for a vehicle system. It starts with an 'Idle' state, which transitions to 'Running' upon 'engineOn'. From 'Running', the system can transition to 'Running' (via 'accelerator' or 'brake'), 'Cruising' (via 'on'), or 'Idle' (via 'engineOff'). From 'Cruising', it can transition to 'Standby' (via 'accelerator' or 'engineOff'), 'Cruising' (via 'brake' or 'on'), or 'Standby' (via 'off'). From 'Standby', it can transition to 'Cruising' (via 'accelerator' or 'resume'), 'Standby' (via 'brake' or 'engineOff'), or 'Idle' (via 'off').

Transition Tree 2 (TT2):

```

graph LR
    Idle1[Idle] -- engineOn --> Running1[Running]
    Running1 -- accelerator --> Running2[Running]
    Running1 -- brake --> Running2
    Running1 -- on --> Cruising1[Cruising]
    Running1 -- engineOff --> Idle2[Idle]
    Cruising1 -- accelerator --> Standby1[Standby]
    Cruising1 -- brake --> Standby1
    Cruising1 -- on --> Cruising2[Cruising]
    Cruising1 -- engineOn --> Standby2[Standby]
    Cruising1 -- engineOff --> Idle3[Idle]
    Cruising1 -- off --> Standby3[Standby]
    Standby1 -- on --> Cruising3[Cruising]
    Standby1 -- engine --> Cruising3
    Standby1 -- accelerator --> Standby2
    Standby1 -- brake --> Standby2
    Standby1 -- engineOff --> Idle4[Idle]
  
```

tion Tree 3 (TT3):

```

graph LR
    Idle1[Idle] -- engineOn --> Running1[Running]
    Running1 -- arbitrary --> Running2[Running]
    Running1 -- brake --> Standby1[Standby]
    Running1 -- on --> Cruising1[Cruising]
    Running1 -- engineOff --> Idle2[Idle]
    Cruising1 -- brake --> Standby2[Standby]
    Cruising1 -- on --> Cruising2[Cruising]
    Cruising1 -- accelerator --> Standby3[Standby]
    Cruising1 -- engineOff --> Idle3[Idle]
    Standby1 -- brake --> Standby4[Standby]
    Standby1 -- engineOn --> Cruising3[Cruising]
    Standby1 -- engineOff --> Idle4[Idle]
    Standby1 -- accelerator --> Standby5[Standby]
  
```

23

diagram is depicted in Appendix B. The methods in `OrdSet` can be classified into groups, such as constructors, add/remove items to/from the set and accessors to class attributes [Antoniol 2002]. For the sake of simplification, the case study is carried out on a subset of `OrdSet`. The subset contains methods that create and modify sets: constructor `OrdSet(int n)`, operator `+=(int n)` and `remove(int n)`. Accordingly, faults are only seeded in the methods relevant to the subset (see Section 3.5.2). The corresponding statechart is illustrated in Figure 5. In order to cover the state dependent behavior of `OrdSet`, we use ordered set of initial cardinality three (i.e., the `OrdSet` initially has three empty slots). The size of the set increases by three when resizing is necessary (when resizing is allowed, i.e., when the number of times resizing has been necessary is below `max_accepted_resizes`, three new slots are added), and resizing is accepted only once (`max_accepted_resizes = 1`).

Because of guard conditions on transitions, some paths are infeasible. We cannot simply traverse the statechart without considering the guard conditions when generating the transition tree. For instance, if the guards are ignored, the sequence of `@Start@OrdSet(int n)@Empty@+=(int n)@PartiallyFilled@+=(int n)@Filled`⁶ may be generated by traversing the statechart. But, with the settings described above, this path is actually an infeasible path. In fact, state `Filled` can only be reached by transition sequences that eventually add three different elements into the set⁷.

⁶ Start is the initial state. @ is used as a delimiter between states/method signatures in transition sequences. A transition is represented using the following format `@source_state@event@target_state`. Transition sequences are therefore of the form `@state1@event1@state2@event2@state3...`

⁷ For example, `@Start@OrdSet(int n)@Empty@+=(int n)@PartiallyFilled@+=(int n)[not(self->includes(n))@PartiallyFilled@+=(int n)[not(self->includes(n))@Filled)`.

Figure 5. Statechart for OrdSet

There are two triggerless transitions in the statechart:

- (1) @Resizing@[_resized_times<=max_accepted_resizes]@PartiallyFilled
- (2) @Resizing@[_resized_times>max_accepted_resizes]@Overflow.

The transition will be taken if and only if the guard is met once Resizing has been entered. Given our setting, transition (1) will be triggered when the fourth element is added into the set, whereas (2) is fired when the seventh element is added. The sequence below is an example of a path made infeasible because of these guard conditions:

```

@Start@OrdSet( int n )@Empty@+=( int n )@PartiallyFilled@
+=( int n ) [ not( self->includes( n ) ) ]@PartiallyFilled@
+=( int n ) [ not( self->includes( n ) ) ]@Filled@
+=( int n ) [ not( self->includes( n ) ) ]@Resizing@
[ _resized_teiems>max_accepted_resizes ]@Overflow
  
```

Yet, another difference with the first case study is that some events in the transitions have parameters. Different actual parameters in test sequences may lead to

different fault detection results. To account for such a source of variation, several instances of the same sequence are included in the test pool. For practical reasons, for each sequence, we randomly select a pre-set number of parameterized sequences (5 in our case study) with distinct actual parameters.

For instance, for the sequence

```
@Start@OrdSet(int n)@Empty@+=(int n)@PartiallyFilled@
+=(int n)[not(self->includes(n))@PartiallyFilled@
remove(int n)[not(self->includes(n))@PartiallyFilled
```

we can have the two following parameterized sequences:

```
@Start@OrdSet(1)@Empty@+=(5)@PartiallyFilled@+=(12)@PartiallyFilled@
remove(10)@PartiallyFilled
```

or

```
@Start@OrdSet(3)@Empty@+=(20)@PartiallyFilled@+=(7)@PartiallyFilled@
remove(5)@PartiallyFilled
```

Although they take the same path in the statechart, but we could include both of them in the test pool. This is the main reason that makes the test pool in the second case study much larger than in the first one.

In order to account for guard conditions and infeasible paths, the transition tree used in the experiment (Figure 6) is generated by adapting Binder's algorithm. If the guard condition is a simple predicate or only contains AND operators, one new tree branch is sufficient. If the guard condition is composed of one or more OR operators, several distinct branches are drawn, one for each truth combination that is sufficient to make the guard true [Binder 1999]. In addition, when the tree construction algorithm does not allow us to derive a feasible path for a given transition, i.e., the path does not lead to fulfillment of the transition guard condition, we derive the shortest path that

allows us to traverse the transition. Our example in Figure 6 shows an additional path that allows the tree to cover the transition from Resizing to Overflow.

Figure 6. Transition Tree (OrdSet)

3.5 Mutation Operators

We used the mutation operators provided in [King 1991, Kim 1999, Kim 2000] to seed faults in the code of the two case studies. Our goal was to cover all the mutation

operators that were applicable in the code under test and to seed the faults in a way that balanced across operators given the characteristics of the code of each case study.

3.5.1 Case Study 1: Cruise Control

As expected, by analyzing the code, some mutation operators turned out not to be applicable⁸ and some operators caused compilation errors⁹ and were therefore not considered. Six mutation operators were eventually used in the case study: Arithmetic Operator Replacement (AOR), Constant Replacement (CRP), Method Name Replacement (MNR), Relational Operator Replacement (ROR), Return Statement Replacement (RSR) and Statement Deletion (SDL).

We seeded 91 faults in three core classes of the system – 28 in class `CarSimulator`, 9 in class `SpeedControl` and 54 in class `Controller`, respectively. The histogram of fault distributions across mutation operators is shown in Figure 7 for all the three control classes. The variation in mutants seeded across classes is justified by the relative complexity of these classes. For example, class `Controller` has more method calls, more methods defined, and a more complex state invariant than `SpeedControl`. The detailed mutant list is given in Appendix E.

⁸ Class Instance Creation Expression change (ICE): No compatible class names; Overriding Method Removal (OMR): No inheritance.

⁹ Exception Handler Removal (HER): We can not remove the exception handlers in `run ( )` sections.

Figure 7. Fault Distributions across Mutation Operators (Cruise Control)

3.5.2 Case Study 2: OrdSet

101 mutants were generated by seeding faults in constructor `OrdSet(int n)` and methods `updateLast()`, `defSetSize(int n)`, `resizeArray()`, `remove(int n)`, `operator +=(int n)`, `make_a_free_slot(int n)`, `binSearch(int *a, int size, int x)`, and `operator ==(int n)`. After analyzing the code under test, eleven applicable mutation operators were used in this case study. In addition to the AOR, MNR and SDL operators mentioned in the Cruise Control case study, Control-Flow Disruption (CFD), Literal Change Operator (LCO), Language Operator Change (LOR), Method Parameter Order Change (POC), Scope Change Operator (SCO), Source Constant Replacement (SRC), Statement Swap Operator (SSO), Variable Replacement Operator (VRO) were also applied. The distribution of mutants among these mutation operators is shown in Figure 8. The detailed mutant list is given in Appendix F.

Figure 8. Fault Distributions across Mutation Operators (OrdSet)

Chapter 4 Results

We used the tool framework described in Chapter 6 to perform the two case studies introduced in Section 3.4. In this chapter, we look at the two case studies in sequence (Sections 4.1 and 4.2) and then draw overall conclusions by identifying common results and explaining differences. For each case study, we report results in the same order, that is in the order the data analysis procedure was introduced in Section 3.3. The first subsections (Sections 4.1.1 and 4.2.1 for the two case studies, respectively) look at the relationships between mutation score and cumulative length (test set size) for the three/four adequacy criteria (All Transitions – AT, All Transition Pairs – ATP, Transition Tree – TT, and Full Predicate – FP when relevant) and compares them to what would be obtained by selecting test cases randomly within the same test set size range. Section 4.1.2 (respectively 4.2.2) compares these relationships and the ones between coverage ratio and cumulative length across the all adequacy criteria, in order to determine what adequacy criterion could be recommended based on our results. Sections 4.1.3 and 4.1.4 (respectively 4.2.3 and 4.2.4) study the mutation score distributions for adequate test sets and perform statistical testing to confirm, in a more formal way, what was visible in previous sections. Section 4.1.5 performs a qualitative analysis of live mutants for the transition tree criterion in the Cruise Control case study in order to better understand the deficiencies of this well-known technique. Section 4.2.5 discusses why certain mutants are not being able to be detected in the OrdSet case study.

4.1 Case Study 1: Cruise Control

4.1.1 Cost-Benefit Analysis (Cruise Control)

Let us first have a look at the relationship between benefit and cost of using coverage criteria, that is between mutation score and cumulative length in our context (Figure 9). Note that we report results for all three possible transition trees shown in Figure 4, i.e., Transition Tree 1 (TT1), Transition Tree 2 (TT2), and Transition Tree 3 (TT3).

From the left column in Figure 9, we can see that the AT and ATP coverage criteria show an expected, monotonically increasing curve and converge towards a 100% mutation score, though at very different size ranges (25 for AT and 100 for ATP). This is, however, not the case of TT threshold and results vary a great deal according to the way the tree is produced. This issue will be clarified when we investigate the reasons why mutants are not killed with the transition trees (Section 4.1.5). Because of the length distributions of the test cases, we do not have observations below a cumulative length of 3 for AT (respectively 7 and 2 for ATP and TTs). Both the splines and the observations are plotted so that it is clearly visible that there is substantial uncertainty around the splines, that is a lot of variation due to randomness in test case selection and their varying effect on mutation scores. This is especially true in the lower part of the cumulative length range.

The right column in Figure 9 plots, for each adequacy criterion, what can be obtained by chance if test cases are randomly selected until the maximum cumulative length of the corresponding adequacy criterion is reached. In short, it models the effect of test set size on the mutation score. With respect to the AT criterion, we can see that the

corresponding null criterion spline is slower to converge towards the maximum mutation score. There is also substantially more uncertainty, that is the spread of observations around the spline is clearly larger. This is what we would expect if a coverage criterion shows more effect on fault detection than a pure test size effect.

With respect to the ATP criterion, the graphical comparison of coverage and pure size effects is less obvious. We can tell that there is much more uncertainty but the splines seem very similar. To be more precise, in Sections 4.1.3 and 4.1.4 we compare the mutation score distributions of the null criterion and every adequacy criterion when coverage approaches 100%.

The bottom part of Figure 9 shows that the null criterion can perform better than the TT1 and TT3 criteria in terms of mutation score. This can be explained by the fact that there are often different ways to reach a particular transition in the statechart. Across the 100 test sets generated, the null criterion uses more ways to do so whereas the paths tested in TT1 and TT3 do not seem to lead to the detection of a number of mutants. This becomes clear when we investigate the reasons why some mutants are not killed with the TTs criteria (Section 4.1.5).

Figure 9. Mutation Score versus Cumulative Length (Cruise Control)

4.1.2 Further Comparison of the Three Coverage Criteria (Cruise Control)

If we now turn our attention to the respective cost of reaching a 100% coverage, Figure 10(a) shows the relative cost of doing so for each of the three criteria. Only the splines are shown so as not to clutter the figure. As expected, we can see that AT is much less expensive than the two others, whereas TT is a good compromise in between the two other criteria in terms of cumulative length. The AT curve starts with a steeper slope as it is easier to find test cases to increase coverage for this criterion so as to converge faster towards full coverage.

Figure 10. Bivariate Relationships (Cruise Control)

From Figure 10(b), we can see that the AT and TT2 curves are very close, thus showing that for similar coverage, they achieve similar mutation scores. TT1, TT2, and TT3 curves have similar shapes until they reach a 60% coverage and then TT1 and TT3 diverge and reach lower mutation scores. Considering that TT criteria have twice the cost of AT for a similar or worse detection effectiveness, TT does not appear to be a cost-effective alternative to AT and ATP.

Figure 10(c) confirms what is said above regarding TT coverage: TT seems to be an inefficient technique for covering transitions. The AT curve converges faster towards the maximum mutation score and reaches a similar or better score than TTs. It also seems to reach a similar mutation score as ATP, but we will see in the next section that this is misleading as with the 100% AT coverage, the risk of missing faults is still high as compared to a 100% ATP coverage. But under certain circumstances where time to market is paramount, it is perhaps conceivable that AT be deemed more appropriate.

4.1.3 Comparing the mutation score distributions of the three Criteria (Cruise Control)

We now compare the mutation score distributions of the three adequacy criteria for all adequate test sets that were generated during the simulation. The five distributions are shown in Figure 11 for AT, ATP, and TTs. They are all based on 100 randomly generated adequate test sets.

Table 1 summarizes the mean and the 2.5% and 97.5% interval quantiles for each distribution. We can see that with ATP, one is almost guaranteed to detect all faults. With AT, however, it is very unlikely one will detect all faults. Regarding TTs, Figure 11 and Table 1 confirm the important variations across the three possible transition trees. With

TT1 and TT3, only 91% and 85% of mutants are killed. However, for TT2, the mutation score mean is close to 1 (0.963) implying one will detect most faults (i.e., TT2 is close to ATP).

Figure 11. Mutation Score Distributions for Adequate Test Sets (Cruise Control)¹⁰

Although the cumulative length of TT adequate test sets is significantly smaller than ATP, we can see that the mutation scores may vary a great deal across transition

¹⁰ The upper part in each distribution diagram is the quantile box which provides a graphical view of the mean, maximum, median, minimum and quantile values.

trees, thus introducing a lot of uncertainty in fault detection rates. This analysis confirms that TT cannot be considered a safe, cost-effective coverage criterion.

Criterion	Mean	2.5% quantile	97.5% quantile
AT	0.9643	0.9136	1.00
ATP	1.00	1.00	1.00
TT1	0.9136	0.9136	0.9136
TT2	0.9630	0.9630	0.9630
TT3	0.8519	0.8519	0.8519

Table 1. Mutation Score Distribution Statistics for all Adequate Test Sets (Cruise Control)

4.1.4 Further Comparing Mutation Score Distributions (Cruise Control)

In this section, we investigate the practical impact of using the coverage criteria presented above. In order to test, using statistical inference tests, whether mutation score distributions from adequate test sets and random test sets of equivalent size would have significantly different means, we follow the following procedure:

- For each adequacy criterion, get the subset of test sets generated that shows a coverage ratio equal or above 90% (Subset S_1) and identify the minimum cumulative length.
- Get the subset of test sets in the Null Criterion simulation that show a size equal or above that minimum cumulative length (Subset S_2).
- Compare the mutation score distributions of S_1 and S_2 and test their differences in means. Recall from Section 3.3 that the results of such a comparison are conservative as with this procedure, subset S_2 tends to have higher average cumulative lengths than subset S_1 .

The rationale is to compare test sets of comparable cumulative length and only consider the high coverage range above 90%, which we would expect to be a minimum to achieve in practice, based on the results in Section 4.1.2. The distributions of S_1 for each criterion are shown in the left column of Figure 12, whereas the right column shows the corresponding Null criterion distributions (S_2).

First, if we compare each mutation score distribution to its corresponding null criterion distribution, a statistical test¹¹ reveals that, for each of the three adequacy criteria, a significantly higher average mutation score can be obtained by driving the selection of test cases using the coverage criterion (see Table 2). A t-test comparing the means of distributions yields a p-value < 0.0001 for each adequacy criterion. This suggests that coverage criteria are practically useful as, for a given test set size, they ensure significantly better mutation scores.

If we now compare the distributions of the three coverage criteria, we see that the ATP yields, on average, higher mutation scores than TT2 (averages: 1.00 vs. 0.9577), which itself yields a similar score to AT (averages: 0.9577 vs. 0.9548). As observed before, TT is at best as effective at killing mutants as AT but significantly more expensive.

¹¹ We used a few parametric and non-parametric statistical tests which yielded consistent results. A t-test yields p-values far below our 0.05 threshold, for all the AT, ATP, and TT criteria.

Figure 12. Distributions of Mutation Scores when Coverage is > 90% (Cruise Control)

Criterion		Mean	2.5% quantile	97.5% quantile
AT	AT	0.9548	0.8765	1.00
	Null	0.8525	0.7037	0.9753
ATP	ATP	1.00	1.00	1.00
	Null	0.9637	0.8969	1.00
TT	TT1	0.8946	0.8272	0.9136
	TT2	0.9577	0.9383	0.9630
	TT3	0.8363	0.7654	0.8519
	Null	0.9142	0.8000	0.9877

Table 2. Mutation Score Distribution Statistics for Test Sets with Coverage > 90% (Cruise Control)

4.1.5 Detailed Analysis of Live Mutants (Cruise Control)

Recall from Section 2.2 that the termination criterion for a transition tree is when a state was already encountered at the same level or above in the transition tree. In this case, from the Cruising state (i.e., once Idle and Running have already been encountered), the breadth first search adds states Standby (three times because of events brake, accelerator, and off), Cruising, and Idle. States Cruising and Idle, which are already in the tree, are then leaf nodes. The three occurrences of state Standby (new state in the tree) produce the three different transition trees we named TT1 (the tree is further developed after event brake, as in Figure 4(a)), TT2 (the tree is further developed after event accelerator as in Figure 4 (b)), and TT3 (the tree is further developed after event off as in Figure 4 (c)).

As discussed before (see Table 1), Transition Tree 1, 2, and 3 do not allow the detection of certain mutants: the mutation scores are on average 0.9136, 0.9630, and

0.8519 respectively. We looked at all the mutants that were missed by at least one TT-adequate test set (for each of the three transition trees) and tried to understand why this was the case. In other words, we identify what sequence of events could have detected the mutant and was not executed in the test set(s).

First, when executing test cases on mutants we both checked the state invariants of the different states traversed, and the corresponding responses, i.e., the value of the speed and brake pedal throttle. A mutant was said to be killed by a test case when either an expected state invariant was broken or a response was incorrect. The non-equivalent mutants missed by the three transition trees are listed in Table 3.

Transition Tree	Alive Mutant #
TT1	8, 16, 24, 25, 26, 59, 69
TT2	63, 64, 71
TT3	4, 8, 16, 22, 24, 25, 26, 59, 63, 64, 69, 71

Table 3. Live mutants for transitions trees (Cruise Control)

What is common to all mutants missed by TT1 is that they can only be detected if the accelerator event is executed before the execution of the method in which the mutant was seeded¹². As an example (the other mutants are described in Appendix G), let us look at mutant #8 that was seeded in method `brake` in class `CarSimulator` according to mutation operator ROR (Relational Operator Replacement): statement `if (throttle>0) throttle=0.0` was changed to `if (throttle<0) throttle=0.0`. The value of `throttle` indicates how hard the driver presses on the gas pedal. In order to detect the mutant, `throttle` must have a non null value before the

¹² Similar observations can be made for TT2 and TT3 (see Appendix G), but with different combinations of events.

execution of brake. However, the only method that increases the value of throttle is accelerator and none of the paths in TT1 executes accelerator before brake (Figure 4(a)). In fact, it appears that using TT1 corresponds to testing the cruise control of a stationary car, though TT1 is a possible transition tree produced by the algorithm described in [Chow 1978, Binder 1999].

Since the statechart for the Cruise Control is not completely specified (Figure 3), we also checked whether testing sneak paths¹³ would have killed these mutants. This analysis showed that testing sneak paths would not improve fault detection in this case. First, if we use built-in test support to force the state and test unspecified transitions, either the test cases do not contain the only events that can reveal faults, that is events accelerator and brake (e.g., testing event off in state Standby), or these two events are used on a stationary car (unspecified transitions for state Idle). If instead, we use sequences of events to reach the states and then test unspecified transitions, it is very unlikely that the test cases will contain sub-sequences that will reveal faults since when reaching a particular state one would devise the shortest sequences possible. As an example, consider state Standby and unspecified transitions engineOn and off. In order to reach Standby, sequence engineOn-on-accelerator from state Idle is sufficient and will not kill additional mutants. However, other possible sequences to reach Standby, e.g., engineOn-on-accelerator-brake, can kill additional mutants. But in this case, additional mutants are not killed because of the test of sneak paths but thanks to the sequences used to reach states.

¹³ In Figure 3, unspecified transitions are events engineOn and off in state Standby, engineOn and resume in state Cruising, engineOn, off and resume in state Running, and every event except engineOn in state Idle.

To conclude this section, we see that it is important to choose carefully the transition tree used to achieve TT coverage, whenever there are alternative choices. In our case study, two of the alternative trees do not exercise common usage scenarios (e.g., exercise cruise control functionalities on a moving car) and lead to poor fault detection. There are several criteria according to which one could select a tree, for example based on expected usage or code coverage, but this issue remains to be investigated.

4.1.6 New Termination Criterion for Transition Trees

As an attempt to finding a compromise between the vastly different AT and ATP criteria, we could slightly modify the termination criterion for building transition trees. To that effect, we propose a new definition: A node (state) in a transition tree is terminal if the state is a final state or has already been encountered at any level in the tree *above* the current level. The transition tree generated following this new rule is shown in Figure 13. This leads, in our example, to kill all the mutants in Table 3 (and thus reaching 100% mutation score) and is achieved at the expense of adding 10 test cases to the test set.

different results in terms of fault detection and cost, thus warranting some intermediary coverage strategy.

Figure 14. Bivariate Relationships with all criteria (Cruise Control)

When analyzing the mutation score distributions of test sets that show a TTM coverage ratio equal or above 90% and compare it to the corresponding null criterion distribution (Figure 15 and Table 4), results clearly indicate that the mutation score means are significantly different ($p < 0.0001$), thus suggesting, like for the other criteria, that the TTM coverage strategy is practically useful. It is also noteworthy that TTM results here are very close to what was obtained with ATP: an average of 0.9912 versus 1.0.

Figure 15. Distributions of Mutation Scores when Coverage is > 90% with TTM (Cruise Control)

Criterion	Mean	2.5% quantile	97.5% quantile
TT - modified	0.9912	0.9630	1.00
Null Criterion	0.9553	0.8395	0.9877

Table 4. Mutation Score Distribution Statistics for TTM Test Sets with coverage > 90% (Cruise Control)

4.2 Case Study 2: OrdSet

4.2.1 Cost-Benefit Analysis (OrdSet)

From Figure 16, we can see that criterion AT and ATP show trends comparable to the first case study. Both curves converge towards a 100% mutation score and the cost of achieving adequate test sets is much more expensive for ATP, about twice as expensive in this case. In both cases, significant variation in mutation score can be observed for a given cumulative length, especially in the lower part of the range. Only at the very end of the cumulative length range, mutation scores consistently yield (nearly) perfect mutation scores. When comparing the AT and ATP curves with their null criterion counterpart, it is clear that the latter shows higher variance in mutation scores and does not converge

towards a 100% mutation score, thus once again demonstrating the usefulness of using such coverage criteria to guide testing and gain confidence in the software under test.

Regarding TT, there is only one possible transition tree (Figure 6) in this case and its cost is comparable to that of AT. Its mutation score is not clearly better or worse than AT either but this is investigated into more detail in the next sections. A comparison with its null criterion counterpart clearly shows more variance in mutation scores for the latter. The FP criterion shows to be even more expensive than ATP but does not show a clear advantage in terms of mutation score. Considering that this case study shows numerous and complex guard conditions, it was to be expected that large number of test cases would be required to fulfill FP. As for other criteria, its null criterion counterpart shows non-convergence towards a 100% mutation score, and a great deal of variance in mutation score.

Figure 16. Mutation Score versus Cumulative Length (OrdSet)

4.2.2 Further Comparison of the Four Coverage Criteria (OrdSet)

If we now compare the bivariate splines across criteria (Figure 17), we clearly see, as for the first case study, that TT is in-between AT and ATP in terms of coverage ratio / test set size relationship (Figure 17(a)). As for the first case study, it takes more effort to fulfill TT than AT¹⁴, and ATP than TT, though the difference between the latter two is significantly larger. It is interesting to note that the FP spline is comparable to that of ATP, though slightly slower to converge and more expensive.

Figure 17(b) shows that for a similar coverage ratio ATP, FP, but also TT provides a similar mutation score, whereas AT yields a significantly lower mutation score. This is different from case study 1 where TT was more comparable to AT. As described in Section 3.4.2, this is due to the fact that TT is a more demanding criterion here as the statecharts has complex guard conditions thus requiring, based on Binder's strategy ([Binder 1999], Chapter 7), that each truth value combination making a guard true be represented by a distinct tree branch. In this situation, TT is closer to FP than it is to AT whereas in the first case study TT could be seen as a specific way to simply cover all transitions.

Figure 17(c) confirms what was observed above by showing that FP tends to be slightly more expensive than ATP, and that AT and TT show adequate test set sizes that are roughly half that of FP and ATP so as to achieve complete coverage.

¹⁴ In terms of the number of test cases in the adequate sets, AT requires an average of 4.27 whereas TT requires 8.

Figure 17. Bivariate Relationships (OrdSet)

4.2.3 Comparing the mutation score distributions of the four Criteria (OrdSet)

If we now look more closely at adequate test sets (Figure 18 and Table 5), we see that mutation scores for ATP adequate test sets are consistent with the first case study: except for a few cases, they reach a perfect mutation score. Similarly, AT shows again to be unreliable as a criterion to ensure high fault detection rates as AT-adequate test sets have a high probability to miss a significant number of faults.

Figure 18. Mutation Score Distributions for Adequate Test Sets (OrdSet)

Criterion	Mean	2.5% quantile	97.5% quantile
AT	0.9520	0.8778	0.9889
ATP	0.9967	0.9556	1.00
TT	0.9740	0.9000	1.00
FP	0.9950	0.9556	1.00

Table 5. Mutation Score Distribution Statistics for all Adequate Test Sets (OrdSet)

One important difference with the first case study, that is consistent with the previous graphs, is that TT adequate test sets show a rather good mutation score distribution, at least significantly better than AT. It is far from being perfect but considering its test set size is similar to the one of AT (Figure 17(a)), it may be a good alternative in practice if ATP is considered to be too expensive given the criticality of the class(es) under test.

On the other hand, FP tends to be more expensive than ATP while providing similar results in terms of mutation score. Based on the results of this case study, which may only be representative of statecharts with numerous and complex guard conditions, it is therefore not advised to use FP. This result has to be confirmed and replicated by subsequent studies.

4.2.4 Further Comparing Mutation Score Distributions (OrdSet)

As for the first case study, in order to confirm, in the statistical sense, that coverage criteria are more beneficial than random test sets of the same size and that effect cannot be reduced to a size effect alone, we compare the distributions of mutation scores for test sets with coverage ratios above 90% and null criterion test sets of similar size (see Section 4.1.4 for further details). A graphical analysis (Figure 19), the analysis of Table 6, and a t-test confirm that, for each criterion, the differences in distributions are practically and statistically significant ($p\text{-value} < 0.0001$). This is consistent with our first case study and confirms such coverage criteria are useful as they lead to significantly better mutation scores for comparable test set sizes.

Criterion		Mean	2.5% quantile	97.5% quantile
AT	AT	0.8855	0.7556	0.9889
	Null	0.7811	0.6333	0.8778
ATP	ATP	0.9933	0.9111	1.0000
	Null	0.8642	0.8111	0.8778
TT	TT	0.9740	0.9000	1.0000
	Null	0.8262	0.6836	0.8778
FP	FP	0.9946	0.9556	1.0000
	Null	0.8693	0.8222	0.8778

Table 6. Mutation Score Distribution Statistics for Test Sets with Coverage > 90% (OrdSet)

Figure 19. Distributions of Mutation Score when Coverage is > 90% (OrdSet)

4.2.5 Detailed Analysis of Live Mutants (OrdSet)

There are nonequivalent mutants that were not detected by some adequate test sets. The mutants missed for ATP, TT and FP by at least one adequate test set are listed in Table 7.¹⁵

Criterion	Live Mutant #
ATP	8, 9, 11, 12, 47, 48, 49, 50, 81, 96
TT	8, 9, 11, 12, 46, 47, 48, 49, 50, 81, 91, 94, 96
FP	8, 9, 11, 12, 96

Table 7. Live Mutants with ATP, TT and FP Adequate Test Sets (OrdSet)

Overall, all the missing mutants listed here can be detected if the actual parameters in the test cases take particular values. For instance, mutant #46 can be killed if the element to be removed from the set is not the first one in the set. Mutant #8 can be detected if the actual parameter in the constructor `OrdSet (int n)` is 3.

A careful analysis of 9 Mutants¹⁶ suggests that when testing operator `+=` and method `remove`, it is important to add/remove at/from different positions in the ordered set, i.e., first, intermediate and last positions. Four mutants (#8, #9, #11 and #12) indicate that testing the boundary values in the constructor is important. Appendix H gives the detailed analysis of these missing mutants.

Note that, since the statechart for the `OrdSet` is completely specified (Figure 5), testing sneak paths is not necessary.

The detailed analysis of the live mutants confirms the conclusion drawn by Antoniol et al [Antoniol 2002], that is that effective fault detection requires carefully

¹⁵ AT is omitted here since it misses a large number of nonequivalent mutants, as discussed above.

¹⁶ #46, #47, #48, #49, #50, #81, #91, #94 and #96.

setting the parameters in the events. Statechart based test strategies combined with other test techniques, such as category-partition, yield better fault detection results.

4.3 Conclusions from the Two Case Studies

The two case studies suggest that AT is probably not a reliable strategy to ensure adequate state-based class testing. This criterion leads to a high probability of missing significant numbers of faults. ATP is effective but extremely expensive. It may be the best choice for particularly critical components though. FP, based on our `OrdSet` case study, does not seem to be cost-effective as it is more expensive than ATP and still about as effective at killing mutants. But this result needs to be confirmed by further studies as our example contains complex guard conditions on transitions and this explains why FP was expensive to achieve¹⁷.

The results obtained with TT are mixed and vary across case studies. The main reason is that statecharts with guard conditions bring TT closer to ATP, whereas when no guard conditions are present TT comes down to a specific way to cover all transitions and is therefore similar to AT, i.e., it is a specific way to achieve AT. The `OrdSet` case study, where several complex guard conditions can be found, results that TT is a rather good, inexpensive alternative if there is no need to achieve high detection rates. It is significantly more effective than AT but about as expensive. In the Cruise Control case study, the results of TT were mixed and overall rather poor. This example showed that

¹⁷ For instance, in transition `@PartiallyFilled@operator+=(int n)@PartiallyFilled`, the guard condition is `[self->includes(n) or (last<_set_size-1 and not(self->includes(n)))]` that has the same truth table as `[self->includes(n) or last<_set_size-1]`. The Full Predicate criterion requires three truth value combinations to be covered. In the second case study, when considering the entire statechart, AT requires covering 12 transitions whereas FP requires covering 33 condition/transition combinations.

not every tree should be expected to perform the same way with respect to mutation scores. While alternative transition trees are possible, one should be careful to select the one that exercises the code in the most realistic and complete manner. This can be done by concurrently checking code coverage and by looking at how realistic are the scenarios exercised.

Chapter 5 Threats to Validity

As for any empirical study that involve testing techniques, the results we obtained are based on a specific sample of faults. Such faults can be based on a particular project or can be seeded. For reasons discussed in the introduction, researchers performing empirical studies are often led to seeding faults. But regardless of the sources of the faults, the generality of the results is threatened by the fact that these faults and their distribution across the system classes may not be representative of all situations. In our case studies, in an attempt to alleviate these problems, we have used an up-to-date (at the time we run the experiments) set of mutation operators for Java and C++ so as to ensure sufficient diversity in the faults we seeded. These faults were also distributed across classes based on their complexity.

Another common threat to validity is due to the fact that results may be specific to the two particular systems we used in the case studies. This can be addressed by replicating the simulation on several systems but results will always, by definition, be specific to a set of systems. In this thesis, to alleviate this problem we have selected case studies of non-trivial complexity from two different kinds of class clusters, i.e., a typical piece of a reactive system (i.e., getting events and sending commands) and a typical storage class cluster (i.e., providing operations to store data into a data structure with specific properties). Comparing these two very different examples and their differences helped shed light on the validity and generality of our results.

Chapter 6 Simulation Tool Framework

A framework called Infrastructure For Testing Research (IFTR) was developed to provide an environment which would perform the experiment automatically and efficiently. IFTR is also designed to be able to be reused and extended to perform the experiment on additional test adequacy criteria, as well as other SUT.

Section 6.1 introduces the system requirements and the overview of IFTR. Section 6.2 presents the system analysis and design in terms of the use case model and the subsystem decomposition. Section 6.3 summarizes the current implementation. Section 6.4 describes how to use IFTR to perform the experiment. Section 6.5 explains how the current implementation can be adapted to carry out the experiment if the statecharts have guard conditions on transitions and parameters in events. Section 6.6 presents a discussion concerning the extension of the current version of IFTR. The limitations of the tool and future improvements are summarized in Section 6.7.

6.1 Requirements for IFTR

Rather than testing programs, the purpose of IFTR is to compare different testing techniques in terms of cost and effectiveness. The primary intention of IFTR is to create an environment to automate the experimental testing research.

According to the experiment design (Section 3.2), IFTR should meet the following functional requirements:

- Load and parse the statechart model of the SUT.
- Analyze test requirements, namely the features, for the selected test criteria.

- Construct the test pool that contains a large number of test cases.
- Build the coverage matrix that records which features are covered by which test cases for each given criterion.
- Check if the test pool is adequate for all selected test criteria.
- Accept additional test cases that are defined manually by the user.
- Acquire test oracles, i.e., the expected outputs of the SUT.
- Generate a set of mutants that will be used in the experiment.
- Execute the test cases in the test pool on each mutant.
- Build the result matrix that records which mutants are killed by which test cases.
- Provide a list of potential equivalent mutants according to the test result.
- Generate *C*-adequate test sets for each given test criterion and output the information about the test set size, coverage ratio and mutation score for each test set.

Figure 20 illustrates an overview of IFTR. The inputs provided by the user consist of the following:

- The UML statechart specifications, which include the definition of the transitions and the infeasible paths, if they exist.
- The list of selected criteria involved in the experiment.
- The mutant list for generating mutants.
- The executable test driver that creates the expected outputs, runs test cases on mutants and records the test results. The test driver should check both the state invariants and the responses during the test.

Figure 20. Overview of IFTR

IFTR produces the *C*-adequate test sets as the results of the test simulation. For each test set generated during the test simulation, IFTR should record the test set size, coverage ratio and mutation score, once a test case has been added into the set. IFTR should also provide some intermediate results – such as the test pool, the coverage matrices and the result matrix – to assist the user in performing the experiment. The output files should be able to be reviewed by the user easily and the simulation results should have the capacity to be imported into most statistical analysis software packages, for example, using the plain text format, with the elements separated by appropriate delimiters. A set of input and output files are defined based on the requirements from the experiment design. A detailed explanation of system inputs and outputs is given in Appendix I.

In addition to the functional requirements, reusability and extensibility of the system, are two very important non-functional requirements which must be considered

during the system design. Although the current experiment consists of comparing four state-based test adequacy criteria, based on two case studies, it can be seen that more test criteria and more examples will be involved in the experiment in the future, such as the comparison of white box testing techniques, for instance. IFTR should possess the capability to allow the new features to be added easily without there being big impacts on the existing implementation.

Several assumptions are made regarding the analysis of criterion features, generating the test pool and obtaining the test oracle for the SUT. First, we assume that the program has been tested sufficiently and can be believed to be correct. This assumption implies that the test oracle can be created automatically by executing the test cases on the original SUT. Second, we classify the UML statecharts depending on the existence of guard conditions on transitions and parameters in trigger events. In its current version, IFTR assumes that the statecharts do not have guard conditions on transitions and no parameters in the events. Generating test cases and analyzing test requirements for statecharts with guard conditions and parameters are research subjects in their own right and is out of the scope of this thesis. However, IFTR provides a flexible and extensible infrastructure so that the experiments performed on the statechart with guard conditions and parameters can be carried out when necessary adaptations are made. Our second case study shows such an example and is described in Section 6.5. However, IFTR can be extended easily, as discussed in Section 6.6.

6.2 System Analysis and Design for IFTR

In this section, the system analysis and design are presented in terms of a use case model and subsystem decomposition.

6.2.1 Use case diagram

Figure 21 shows the use case diagram for IFTR. There are 13 use cases in total and the use cases are invoked based on the functionality selected by the researcher when the entry conditions are met.

Figure 21. Use case diagram for IFTR

Researcher is the actor in the system. Researcher initiates a request to perform the experiment and provides the SUT and the corresponding statechart specifications, the mutant list that defines the faults needing to be seeded. The statechart is loaded into the system by use case Load Model. Use case Generate Mutants

creates a number of mutants, according to the mutant list. The test requirements for the given criterion, i.e. the criterion features, are identified by use case Analyze Features. A test pool is then constructed by invoking use case Construct Test Pool. The coverage matrices are built by use case Build Coverage Matrices. Researcher needs to check the coverage matrices and decide whether the test pool is sufficient for all selected criteria, and whether additional test cases are necessary. If additional test cases need to be added into the test pool, use case Add Test Cases and Prune Test Pool are then invoked. Generating the test pool is an iterative process. The coverage matrices used by the test simulation have to be created based on the final test pool.

Researcher must implement a test driver that can obtain the expected outputs for the SUT, and record the actual outputs and the testing result when running the test cases on the mutant set. Use case Create Oracle executes the test driver to acquire the expected outputs for testing the SUT. The test pool can be executed on the mutants by use case Run Test once the test oracle has been created. Use case Build Result Matrix generates the result matrix. Researcher determines the equivalent mutants in the potential equivalent mutant list provided by use case Identify Equivalent Mutants. Use case Perform Simulation can then be invoked to generate test sets. Once the simulation is completed, use case Analyze Data is invoked to analyze the result data.

The description of the use cases in terms of the required entry condition, flow of events and expected exit condition is presented in Appendix J.

6.2.2 Subsystem decomposition

By considering improving the cohesion within a subsystem and loosening the coupling between subsystems, the system is decomposed into six subsystems, as shown in Figure 22. The public classes in each subsystem and the associations between these classes are shown to illustrate the organization of the subsystems.

The IFTRGUI subsystem represents the system interface with users. The ModelTransformation subsystem parses the statechart model and analyzes the test criterion features. The MutantGeneration subsystem generates mutants. The TestCaseGenerationExecution subsystem constructs the test pool, executes the test cases and creates the coverage matrices and the result matrix. The TestSimulation subsystem generates test sets for selected test criteria and computes the coverage ratios and mutation scores based on the coverage matrices and the result matrix. The DataAnalysis subsystem performs the statistical analysis on the result data and plots the data in diagrams.

Class GetStatechart, LoadFeatures, GetSUT, LoadMatrix and LoadResultChart are the boundary classes that represent the interfaces between subsystems. The purpose of using boundary classes is to reduce the coupling between subsystems, so as to minimize the impact of changing the design of a subsystem.

The DataAnalysis subsystem will employ statistical software packages to perform the data analysis. JMP, a statistical discovery software developed by SAS Institute Inc., is used in our experiment. Users can select other software packages depending on their preference.

Figure 22. Subsystem decomposition for IFTR

The following sections present the object design details for each subsystem. Our discussion will be focused on the functionalities and class structures of the subsystems. Therefore, in the UML class diagrams depicted in the following subsections, the role and association names have been omitted. Furthermore, to avoid over-cluttering the class

diagram, only the attributes and methods relevant to the explanation of the design are shown.

6.2.2.1 IFTRGUI subsystem

The IFTRGUI subsystem provides the GUI to allow users access to the system. It is composed of class `SystemMenu`, `CommonPanel`, and `UserGuide`. This subsystem also includes several classes that implement the graphic user interface used by other subsystems, such as `FileSelector`, which accepts the selection of input files, and `CriterionSelector`, which allows the user to select desired test criteria. The implementation of the GUI is separated from the model objects. Therefore, a change in the GUI implementation will not affect the underlying model objects. The class diagram is depicted in Figure 23. Note that only the key methods in the `SystemMenu` class are listed and the GUI components, such as `Jpanel`, `JButton` and `JMenu`, are omitted in the diagram.

Class `SystemMenu` extends the `JFrame` class in Java Swing package and provides a set of menu items. When a menu item is selected, the corresponding control object is created and the appropriate methods are invoked. For instance, the `requestLoadModel` and `requestAnalyzeFeature` method create the `ModelControl` object in the `ModelTransformation` subsystem. Class `CommonPanel` implements a GUI that accepts the selection of input files and gives prompt messages, which show whether the process is successful or whether any exception has occurred. Class `UserGuide` displays the user manual in a separate frame.

Figure 23. IFTRGUI subsystem class diagram

6.2.2.2 ModelTransformation subsystem

The ModelTransformation subsystem is responsible for loading and parsing the statechart model, and analyzing the features for the selected criteria. Information about states, transitions, trigger events, guard conditions and actions are first obtained by parsing the statechart model, and are then used during feature analysis and test case generation.

The ModelControl class is created to handle the process of loading and parsing the statechart model, and analyzing criterion features. It collects the user selection of the model file and criteria and then dispatches the selection to the corresponding entity classes.

Figure 24. ModelTransformation subsystem class diagram

With the intention of developing a framework, we apply the variability analysis to identify the variations in model transformation and generation of criterion features. First, the representation format of the statechart model may vary. It could be the plain text format of a transition table, a UML statechart diagram created by certain software such as Rational Rose, or other formats¹⁸. Second, the parsing algorithms differ with respect to the format of the model. Third, the algorithms for analyzing criterion features vary since

¹⁸ We chose transition table as the statechart model for simplicity.

the test requirements of each criterion are different. Based on the variability analysis, the Bridge pattern and the Template Method pattern have been chosen to implement the framework. The classes involved in the design patterns and the roles of the classes are illustrated in Figure 24.

The Bridge pattern decouples a set of implementations from the set of objects using them [Gamma 1995]. The abstractions and implementations are separated into two class hierarchies and can be extended independently. As the class diagram shows in Figure 24, class Model is the abstraction in the Bridge pattern, whereas class Parsing is the implementor. Class Statechart is used as the concreteImplementator that creates the State and Transition objects during the parsing of the statechart. By applying the Bridge pattern, the model of the SUT could be extended to the statechart models other than a transition table, and the parsing of the model might also be implemented differently, depending on the representation format that the model takes. The Bridge pattern allows us to extend the model abstraction and the parsing implementations without the impact on clients.

The Template Method pattern is used for analyzing the test criterion requirements. The Template method consists of an abstract class that defines the basic TemplateMethod which needs to be overridden. Each concrete class derived from the abstract class implements a new method for the template [Shalloway 2002]. As the class diagram shows in Figure 24, class CriterionFeature is the AbstractClass, whereas the AllTransitions, AllTransitionPairs and TransitionTree class are the ConcreteClasss. The getFeatures method in class CriterionFeature is the Hook method that contains the common operations of feature analysis, such as

getting the Statechart object and writing features into the appropriate file. When it calls the abstract method `doAnalyzeFeatures`, the concrete method in the corresponding derived class, which analyzes the features for a given criterion, is invoked. For example, when the user selects an analysis of the test features for the Transition Tree criterion, the `doAnalysisFeatures` method in class `TransitionTree` is invoked. The control is then returned to the `getFeatures` method in class `CriterionFeature`. The Template Method pattern provides a good platform that allows the algorithm to be refined without impacting on other classes. With this design, new test criteria can be added into the system by implementing new derived classes, using the Java dynamic binding feature. The system can be extended without changing the structure of classes.

6.2.2.3 MutantGeneration subsystem

The `MutantGeneration` subsystem is responsible for generating mutants based on the mutant list file created by the user. The class diagram is shown in Figure 25.

We considered using the preexisting mutant insertion tool MIAT, developed by a former graduate student, to generate mutants. MIAT creates mutant programs automatically, based on the mutant information provided by the user. Since the interface of the preexisting class has to be converted to the desired interface, the Adapter pattern is chosen to implement the interface conversion. Class `MutantControl` is the Client class that requests the generation of mutants. It doesn't need to know which tool will be used. Class `MutantGeneration` is the Target class that encapsulates the Adapter classes. Class `MutantGenerationAdapter` is the Adapter class that implements the

generateMutant method and converts the interface of the preexisting tool. The MutantGenerationTool is the Adaptee class that will be adapted into the system.

Figure 25. MutantGenerationElimination subsystem class diagram

6.2.2.4 TestCaseGenerationExecution subsystem

The TestCaseGenerationExecution subsystem is responsible for constructing the test pool, pruning it, obtaining the test oracle and executing all test cases on the mutant set. The coverage matrices and the result matrix are built as the results and will be used by the TestSimulation subsystem.

As in the class diagram shown in Figure 26, class TestPoolControl and TestControl are the control classes in the subsystem. The TestPoolControl class

handles the operations of generating test cases, pruning the test pool and creating the coverage matrices. The test cases are generated by randomly traversing the statechart diagram. Each test case is a path in the statechart diagram that contains a sequence of transitions, which starts from the initial state. The duplicate test cases are identified and removed automatically during the pruning operation. The test cases in the final test pool are distinct from one another. After the test pool has been finalized, the `TestPoolControl` class creates the coverage matrix for each criterion. The `TestControl` class handles the process of executing test cases, creating a test oracle and generating the result matrix. The oracle is created by executing the user-defined test driver that runs the test cases on the original version of the SUT. The `TestControl` class then executes the test driver which runs the test cases on mutants, comparing the outputs with the test oracle to determine which mutants are killed by a particular test case. The results are then recorded in the result matrix.

The boundary class `GetStatechart` and `GetSUT` acquire the information of the statechart and the SUT from the `ModelTransformation` subsystem and the `MutantGeneration` subsystem, respectively. Class `TestOracle` obtains the expected test outputs from the test oracle file created by the user-defined test driver. The `ConfigureTestPool` is a user interface class that allows the user to set the stop condition and maximum length of the test cases for constructing the test pool, and then passes the inputs to the `TestPoolControl` object. The maximum length is the maximum number of transitions allowed in a test case (see Section 3.2). The stop condition is used to terminate the process of generating test cases by tracing the improvement of the test pool during the test case generation. For example, if the user

inputs 10 as the stop condition, the test case generation will be stopped if the number of new distinct test cases (not duplicate with the existing test cases in the test pool) gained in 1000 traverses of the statechart diagram is less than 10. The user can adjust the combination of the maximum length and the stop condition in order to obtain a test pool with a size sufficient for the experiment.

Figure 26. TestCaseGenerationExecution subsystem class diagram

Inheritance and dynamic binding are applied in building up the coverage matrices. Because of the variations with respect to how to assess the coverage of the given criterion, the `setEntries` method in class `CoverageMatrix` is implemented so as to fit the definition used by most of the selected criteria, and it may be overridden by the one in its subclass. For instance, for All Transitions (AT) and All Transition Pairs (ATP), a feature is covered if the test case contains the feature; but for Transition Tree (TT) criterion, a path in the tree is covered if, and only if, the test case is the same as the path. Thus method `setEntries` has to be implemented differently for AT, ATP and TT. The `setEntries` method in class `CoverageMatrix` is implemented by following the coverage definition in AT and ATP to reduce code duplication. With such a design, new criteria can be added without impacting on other classes.

Class `NewTestCase` assists the user in generating additional test cases to improve the coverage of the test pool. This is done by adding a set of test cases that covers the features for the given criteria. For some test criteria, such as Transition Tree, the paths in the tree all start from the initial states, and therefore can be added into the test pool directly. But for criteria like All Transition Pairs, the features may not always start from the initial state, so certain transition sequences have to be added in front so as to convert the features to test cases that start from the initial state. This may become more complicated when there are guard conditions on the transitions and parameters in the events in the statechart. Therefore, the `NewTestCase` class is extended by a set of subclasses, one class per a given criterion, to ensure that the functionality can be extended easily.

6.2.2.5 TestSimulation subsystem

The TestSimulation subsystem is responsible for simulating the test based on the coverage matrices and the result matrix. It generates test sets by randomly selecting rows from the matrices and then computes the coverage ratio and mutation score for each test set. It is also responsible for determining if the number of test sets meets the requirement in the experiment design, i.e. 100 C-adequate test sets for each selected criterion. The results are recorded in simulation result files and will be used by the DataAnalysis subsystem.

Figure 27. TestSimulation subsystem class diagram

The class diagram is shown in Figure 27. The boundary class LoadMatrix acquires the coverage matrices and the result matrix from the TestCaseGenerationExecution subsystem. Thus the TestSimulation subsystem will not be affected by implementation changes in the TestCaseGenerationExecution subsystem. Class ConfigureSimulation allows the user to select criteria and set the number of test sets which need to be generated.

Common operations, such as getCoverageMatrix, getResultMatrix, computeCoverageRatio and computeMutationScore, are implemented in class SimulationControl. Class Phase1Control extends SimulationControl and implements the actual generation of test sets. The reason for designing a separate class called Phase1Control is that it is possible to perform other simulations in the future. The name Phase1 stands for our current design of the experiment. When further design of the simulation takes place later on, it will be named Phase2 and the system can be extended without a big impact on the existing implementation. The subclasses may override the method in class SimulationControl if necessary.

A set of subclasses, one per a given criterion, extends class Phase1Control. This is because different ranking algorithms may be used between adequacy criteria for generating test sets (see Section 3.2). For instance, for AT, the test cases in the test pool are ranked according to their normalized increase in transition coverage which is defined to be the increase of transition coverage a test case yields divided by the length of the test case. A randomly selected test case from those that have the highest normalized increase in coverage is added to the test set. However, for TT, the coverage ratio is assessed in two different ways. One is the coverage of the tree paths and another is the coverage of the

transitions in the tree. The first is used for calculating the coverage ratio of the test set, whereas the second is used for ranking the test cases. This is consistent with the attempt to converge as fast as possible towards full coverage. The most commonly used ranking algorithm is implemented in class `Phase1Control` to reduce code duplication.

6.3 Implementation

IFTR is implemented in Java, a platform-independent programming language. The system is composed of 5 packages, 62 classes and 5500 lines of code (excluding the MIAT tool and comments). The GUI of IFTR is created using Java Swing components. To compile and execute IFTR, JDK 1.4 provided by Sun Microsystems Inc. is required. This is because the `java.util.regex` package that includes the classes for matching character sequences against patterns specified by regular expressions is used in the implementation for handling infeasible paths.

IFTR is multi-threaded. The control classes are implemented by extending the `Thread` class in Java. New threads are created separately from the GUI when the control objects, such as `TestPoolControl` and `TestControl`, are instantiated. The purpose of such an implementation is to coordinate with the GUI for displaying prompts and to allow the user to terminate some time-consuming operations, such as testing and the test pool construction, as needed.

6.4 Usage

IFTR is accompanied by a detailed user guide that describes how to use IFTR. This section gives a brief introduction to how to use the tool.

Figure 28 shows the layout of the system menu frame. The frame consists of a menu bar, an input area and a message box. The menu bar consists of four submenus and two menu items. The user can browse the user guide by clicking on **HELP**, and exit the system by clicking **EXIT**. Most menu items in the subsystem are named after the corresponding use cases and can be selected when the entry conditions are met.

Figure 28. System Menu GUI of IFTR

A set of input parameters is required to activate the system menu. The first parameter, **Model File**, indicates the location of the file that defines the transition table for the SUT. The user then needs to identify the path of the criterion list file that provides a list of the test criteria involved in the case study. The **Project Path** indicates the directory containing the SUT. The **Test Driver File** is the executable test driver written by the user. The name of the initial state and the final state in the statechart need to be inputted for the purpose of generating the test pool. These parameters can be inputted either manually, from the keyboard, or by clicking on the file selector button to browse the directories on the disk. Clicking the **Reset** button can clear these inputs, and clicking the **OK** button will activate the system menu. The **Stop** button allows the user to terminate time-consuming processes, such as generating the test pool and executing the

test pool on mutants, at any time they wish. The message box displays the execution status and warnings if an exception occurs.

The **Model Transformation** submenu provides with the user three choices (Figure 29). (1) **Load Statechart Model** loads and parses the transition table to generate the state and transition objects. (2) **Analyze Feature** creates the criterion features for the selected criteria. (3) **Find Longest Path Length** provides length information about the paths that cover all the states or all the transitions in the statechart, without repetition of transitions. A separate dialog box (Figure 30) pops up once the user has selected **Analyze Feature**, which allows the user to select desired criteria by putting check marks besides the criterion names. The criterion names are obtained from the criterion list file and items in the table will always be consistent with those in the file.

Figure 29. Model Transformation submenu

Figure 30. Select Criterion dialog box

The Mutant Generation submenu (Figure 31) brings the user to the pre-existing mutant generation tool – MIAT. The user needs to identify the mutant list file and the location of the original version of the SUT, and where the mutant programs will be created. The MIAT user guide may be referred to for details.

Figure 31. Mutant Generation submenu

The Test Case Generation & Execution submenu consists of eight items (Figure 32), in which Add Test Cases is a cascaded menu that allows the user to add additional test cases in two different ways. When the user selects Construct Test Pool to produce the test pool, a dialog box will pop up and ask for inputs regarding the maximum length of the test sequence and the stop condition (Section 6.2.2.4). Prune Test Pool ensures that the test

cases in the test pool are unique. Selecting **Build Coverage Matrix** will create the coverage matrices for the selected criteria. Once the test pool has been finalized and the user-defined test driver has been provided, the user can select **Create Test Oracle** to execute the test driver on the original SUT to get the expected values. After the test oracle has been obtained, **Run Test** executes the test cases on mutants. **Build Result Matrix** can be then selected to generate the result matrix based on the test results. **Identify Equivalent Mutant** will provide a list of mutants that were not killed by any test case in the test pool.

Figure 32. Test Case Generation Execution submenu

Test Simulation submenu (Figure 33) provides two options for the user. The user needs to indicate how many test sets have to be generated per criterion after selecting **Generate Test Sets**. If the user wants to investigate the possible subsumption relationships among the given criteria, he or she can choose **Check Subsumption** to compare the criterion features.

Figure 33. Test Simulation submenu

6.5 Carrying out the experiment when the statechart has guard conditions and parameters

The test pool generation and criterion feature analysis are tightly coupled with the characteristics of the statechart. In Section 6.1, we stated that IFTR made an assumption regarding the statechart. With the current implementation of IFTR, the experiment process can be mostly automated if, and only if, the statecharts do not have guard conditions on transitions and parameters in events. However, IFTR can be adapted to handle the statechart with guard conditions and parameters. This section describes the adaptation briefly based on our second case study – Ordset.

The original statechart for OrdSet contains several guard conditions on transitions and parameters in events, and is illustrated in Figure 5. This statechart is transformed into an equivalent statechart with the intention of removing the guard conditions in the statechart. This transformation is based on the settings used for the ordered set, in which the initial cardinality is three, the size of the set increases by three when resizing is possible, and resizing is allowed only once (see Section 3.4.2) In the equivalent statechart

(Figure 34), the states show the number of elements in the ordered set explicitly, and the guard conditions are implicitly represented by the source state and the target state of transitions. For example, transition

```
@PartiallyFilled@remove(int n)[_last=0 and self->includes(n)]@Empty
and
```

```
@PartiallyFilled@+=(int n)[_last=_set_size-1 and not(self->
includes(n))]]@Filled
```

in the original statechart are transformed to transition

```
@PartiallyFilled1@remove1@Empty
```

and two transitions

```
@PartiallyFilled2@+=2@Filled3 and @PartiallyFilled5@+=2@Filled6 in the
equivalent statechart, respectively.
```

For a given adequacy criterion C , the coverage of C may have different meanings in Figure 5 and Figure 34. For instance, covering all transitions in Figure 5 is not equivalent to covering all transitions in Figure 34. However, this can be managed by mapping between coverage in the two equivalent statecharts. This mapping depends on whether the criterion considers the guard conditions. If it does not take the guard conditions, such as AT and ATP, into account, the original statechart should be used for defining the test requirements. Otherwise, when guard conditions are taken into account such as TT and FP, the equivalent one should be used.

Figure 34. The equivalent statechart for OrdSet

A program which considers the guard conditions and parameters in the statechart when constructing the test pool needs to be implemented independently. In addition to generating test cases by randomly traversing the equivalent statechart, this program also has to implement the mapping between the two equivalent statecharts. The test cases in the test pool need to be defined with the naming set in the two equivalent statecharts. The user has to select the appropriate test pool to create the coverage matrices. For instance, AT and ATP do not consider the guard conditions, and therefore the test pool which uses

the naming set in the original statechart is used to build the coverage matrix. For the TT and FP, the test pool which uses the equivalent statechart, is used for generating the coverage matrix. Once the coverage matrices and the result matrix have been acquired, the subsequent procedures can be performed automatically, using IFTR.

6.6 Extending IFTR

IFTR takes a framework approach to achieve sufficient extensibility and reusability. Although the current version of IFTR makes several assumptions (Section 6.1) regarding the SUT, it can be extended without major impact on the existing implementation.

The design combines different software development techniques. A set of intermediate files is created to provide useful information to the user at each step. This also allows the new components can be plugged into IFTR easily. On condition that the predefined formats of these files are met, the user can use IFTR system or part of IFTR to perform the experiment. Moreover, certain software design patterns are used as the building blocks for the framework. In this section, we use several examples to show how IFTR can be extended.

Example 1: The statechart is represented in a format other than a transition table.

There are two ways to include the new format of statechart specifications. One solution is to implement a program separately to parse and convert the format to a transition table that has been already included in IFTR. Another solution is to add a subclass that extends the abstraction class `Model` in the `ModelTransformation` subsystem, and modify the `loadModel` method in class `ModelControl` to instantiate an object of the added subclass.

Example 2: A new state-based criterion need to be involved in the experiment.

Three classes need to be implemented and added into the tool. (1) A concrete class that extends class `CriterionFeature` in the `ModelTransformation` subsystem to implement the `doAnalyzeFeatures` method based on the definition of the test criterion. (2) A subclass that extends class `CoverageMatrix` in the `TestCaseGenerationExecution` subsystem to override the `setEntries` method if necessary. (3) A subclass that extends class `Phase1Control` in the `TestSimulation` subsystem to override the `generatePhase1TestSets` method is required. Once the new classes have been implemented, the user needs to update the criterion list file by adding the name of the new criterion to it.

Example 3: Be able to handle guard conditions in the statechart, assuming one has an algorithm to produce test cases from a statechart with guards.

A subclass that extends class `TestPoolControl` needs to be added to the tool. This subclass implements the algorithm in method `generateTestCases` that overrides the one in `TestPoolControl`. Modifications also have to be made to the relevant classes in the `ModelTransformation` subsystem and the `TestSimulation` subsystem if the guard conditions have an effect on generating criterion features and test sets. In such a case, the steps in Example 2, described above, should be taken.

Example 4: Other testing techniques, such as white box testing based on the code coverage, are involved in the experiment.

In the experiment process design (Section 3.2), a relatively large portion of the experiment design is not restricted within the statechart-based test criteria. In the simplest case, for example, once the coverage matrices and the result matrix have been built, the

subsequent steps of test simulation and data analysis are identical, regardless which test criterion is selected.

The system design provides a flexible platform which allows users to perform the experiment as they wish. Two approaches can be taken to extend the tool. The first is to develop an independent tool that generates the test pool, the coverage matrices and the result matrix, following the file format convention used in IFTR. The user can then plug these files into IFTR to use the test simulation functions provided by IFTR. The second approach implements several classes by following the design patterns used in the subsystem design. In the ModelTransformation subsystem, one needs to implement a RefinedAbstraction class and a ConcreteImplementor class in the Bridge pattern, based on the model required by the testing technique. A new ConcreteClass in the Template Method pattern must also be coded. Moreover, new derived classes have to be added to extend the CoverageMatrix class in the TestCaseGenerationExecution subsystem, as well as the Phase1Control class in the TestSimulation subsystem. Obviously, the latter approach requires a much deeper understanding of the existing implementation.

6.7 Summary

IFTR provides a framework to automate the experiment procedures in an efficient way. Users can make use of the system, or part of the system, to perform an empirical study that compares the cost effectiveness of adequacy criteria. The system relies on inheritance and dynamic binding for extensibility and is implemented with design patterns. Therefore it is easy to extend and reuse. However, IFTR's functionality needs to be improved in some places. There are at least three directions which could be taken for

the future work. First, the tool should be able to parse statecharts defined in the forms other than transition tables, such as XML¹⁹ Metadata Interchange Format (XMI), or integrate with other CASE tools such as Rational Rose. Second, IFTR should be able to construct a test pool and generate coverage matrices for statecharts that have guard conditions on transitions and parameters in events. The latter is more complicated because it depends on the context of the program under investigation. Third, more test criteria should be included in the tool in the future.

¹⁹ XML stands for eXtensible Markup Language.

Chapter 7 Conclusions

This thesis has investigated four of the most referenced adequacy criteria for testing classes based on UML statecharts. It is notorious that adequacy criteria cannot, in general, be compared in terms of cost and fault detection through analytical means [Weyuker 1991]. This thesis proposes here a simulation procedure to empirically investigate the cost benefit of using test case selection strategies based on statecharts. A tool framework has been built to automate this procedure and applied to two different, carefully selected state-dependent class clusters. We analyzed the results in terms of the size of test sets and their fault detection effectiveness.

From the results of our case studies we can draw practical conclusions regarding the suitability of the four adequacy criteria considered: All transitions (AT), All Transition Pairs (ATP), all paths in the statechart Transition Tree (TT) [Binder 1999] and Full Predicate (FP) [Offutt 1999b]. It seems likely that in most practical cases, an AT adequacy coverage will not be sufficient to ensure an adequate level of fault detection. Indeed, it is very unlikely that all or even most faults will be detected if using such an adequacy coverage criterion. On the other hand the ATP seems to offer a very strong guarantee (though no certainty) that (nearly) all faults will be detected. But this comes at an enormous increase in cost, roughly 5 and 3 times the cost of achieving full AT coverage in our two case studies, respectively. Following the technique proposed by Chow [Chow 1978] and adapted to statecharts by Binder [Binder 1999], we can generate so-called transition trees based on analyzing the class statechart. However, our results show that this well-known technique, which has mostly been applied in the context of

protocol testing, is not always cost-effective in the context of class (cluster) testing. Its cost and effectiveness depends on the extent to which guard conditions are present in the statechart. When the statechart does not have guard conditions, TT is essentially a way to cover all transitions and there are less expensive ways to do so following, for example, the procedure we used for AT: After a pool of test cases is automatically generated, test cases are selected in a stepwise manner to maximize the delta transition coverage at each step while minimizing the additional testing cost. However, because AT and ATP criteria entail widely different fault detection ratios and costs, we modify the TT criterion in order to provide a compromise between AT and ATP. Our modified transition tree strategy yields fault detection results that are very close to those of ATP but its cost is significantly lower than that of ATP. Though these results need to be confirmed by further experimentation, it is likely that such a compromise would fit many practical situations. When guard conditions are present, FP does not appear very cost-effective as it is more expensive than ATP but less effective. TT, on the other hand, appears to be a good compromise between AT and ATP as transitions with guard conditions may be traversed several times under different conditions if they contain disjunctions.

Another, perhaps more fundamental question, that we investigate is whether the effect of coverage on fault detection is mostly due to a size effect, that is to the fact that higher coverage test sets are also larger test sets in terms of method invocations and test cases. In other words, would the random selection of test cases (which is much easier and less costly to perform) yield comparable results? With respect to AT, it is rather clear that covering all transitions is a rather basic requirement if one wants to have a reasonable chance to identify most of the faults. However, whether the coverage of all transition

pairs or all (modified) transition tree paths lead to better fault detection and this in addition to the effect of running larger test sets is a question to be investigated. Our results suggest indeed that coverage has, for all the four investigated coverage criteria, an additional, significant impact to test set size with respect to increasing fault detection.

As for any empirical work, our results have of course to be confirmed by additional studies. This is why this thesis paid particular attention to define a precise simulation and experimental procedure that can be replicated. Future work should also include the investigation of selection strategies for building transition trees. When building such trees, there are always a large number of choices regarding how the statechart can be traversed and the tree can be built. Heuristics to select branches that are more likely to detect faults should be investigated. Furthermore, the simulation environment we developed needs to be extended to automate the analysis of statecharts with guard conditions and event parameters. Such simulation capabilities are key instruments for gaining insight into the cost-effectiveness of testing strategies.

References

- [Antoniol 2002] G. Antonioli, L. C. Briand, M. Di Penta and Y. Labiche, "A Case Study Using the Round-Trip Strategy for State-Based Class Testing," *Proc. (to appear) 13th IEEE International Symposium on Software Reliability Engineering*, Annapolis (MD, USA), November 12-15, 2002.
- [Binder 1999] R. V. Binder, *Testing Object-Oriented Systems - Models, Patterns, and Tools*, Addison-Wesley, 1999.
- [Chow 1978] T. S. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Transactions on Software Engineering*, vol. SE-4 (3), pp. 178-187, 1978.
- [Eubank 1988] R. L. Eubank, *Spline Smoothing and Nonparametric Regression*, Marcel Dekker, 1988.
- [Frankl 1993] P. G. Frankl and S. N. Weiss, "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing," *IEEE Transactions of Software Engineering*, vol. 19 (8), pp. 774-787, 1993.
- [Frankl 1997] P. G. Frankl, S. N. Weiss and C. Hu, "All-Uses versus Mutation Testing: An Experimental Comparison of Effectiveness," *Journal of Systems and Software*, vol. 38 (3), pp. 235-253, 1997.
- [Gamma 1995] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

- [Hong 2000] 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.
- [Hutchins 1994] M. Hutchins, H. Froster, T. Goradia and T. Ostrand, "Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria," *Proc. 16th IEEE International Conference on Software Engineering*, Sorrento (Italy), pp. 191-200, May 16-21, 1994.
- [JMP 2000] "JMP Statistics and Graphics Guide," SAS Institute, V. 4, 2000.
- [Kim 1999] S. Kim, J. A. Clark and J. A. McDermid, "The Rigorous Generation of Java Mutation Using HAZOP," *Proc. 12th International Conference on Software & Systems Engineering and their Applications (ICSSEA'99)*, CNAM - Paris, France, pp. 9-10(11), December 8-10, 1999.
- [Kim 2000] S. Kim, J. A. Clark and J. A. McDermid, "Class Mutation: Mutation Testing for Object-Oriented Programs," *Proc. Net.ObjectDays*, Erfurt, Germany, October 9-12, 2000.
- [King 1991] K. N. King and A. J. Offutt, "A Fortran Language System for Mutation-Based Software Testing," *Software - Practice and Experience*, vol. 21 (7), pp. 686-718, 1991.
- [Kung 1994] D. Kung, N. Suchak, J. Gao, P. Hsia, Y. Toyoshima and C. Chen, "On Object State Testing," *Proc. Proc. of COMPSAC'94*, pp. 222-227, 1994.
- [Lee 1996] D. Lee and M. Yannakakis, "Principles and Methods of Testing Finite State Machines - A Survey," *Proceedings of the IEEE*, vol. 84 (8), pp. 1090-1123, 1996.

- [Magee 1999] J. Magee and J. Kramer, *Concurrency, State Models & Java Programs*, Wiley, 1999.
- [Mathur 1994] A. P. Mathur and W. E. Wong, "An Empirical Comparison of Mutation and Data Flow Based Test Adequacy Criteria," *Journal of Software Testing, Verification, and Reliability*, vol. 4 (1), pp. 9-31, 1994.
- [Offutt 1999a] A. J. Offutt and A. Abdurazik, "Generating Tests from UML specifications," *Proc. 2nd International Conference on the Unified Modeling Language (UML'99)*, Fort Collins, CO, pp. 416-429, October, 1999.
- [Offutt 1999b] A. J. Offutt, Y. Xiong and S. Liu, "Criteria for Generating Specification-Based Tests," *Proc. 5th International Conference on Engineering of Complex Computer Systems (ICECCS)*, Las Vegas, NV, pp. 119-129, October, 1999.
- [Shalloway 2002] A. Shalloway, J. R. Trott, *Design Patterns Explained*, Addison-Wesley, 2002
- [Weyuker 1991] E. Weyuker, N. Stewart, S. N. Weiss and D. Hamlet, "Comparison of Program Testing Strategies," *Proc. ACM International Symposium of Software Testing, Analysis and Verification*, Victoria (BC, Canada), pp. 1-10, October 8-10, 1991.
- [Weyuker 1993] E. Weyuker, "More Experience with Data Flow Testing," *IEEE Transactions on Software Engineering*, vol. 19 (9), pp. 912-919, 1993.

Appendix A Class Diagram for Cruise Control System

The Cruise Control System in [Magee 1999] is selected as the first case study in our experiment. The system is implemented in Java and composed of six classes. The class diagram is shown in Figure 35.

Figure 35. Class diagram for Cruise Control System

Appendix B Class Diagram for OrdSet

The OrdSet is selected as the second case study in our experiment. The program is implemented in C++ and composed of three classes. The class diagram is shown in Figure 36.

Figure 36. Class diagram for OrdSet

Appendix C Coverage Ratio versus Cumulative Length (Cruise Control)

Figure 37. Coverage Ratio versus Cumulative Length (Cruise Control)

Appendix D Coverage Ratio versus Cumulative Length (OrdSet)

Figure 38. Coverage Ratio versus Cumulative Length (OrdSet)

Appendix E Mutants Designed for Cruise Control Case Study

This appendix provides the detailed information about the faults seeded for Cruise Control case study.

Mutant #	Filename	Line #	Original statement	Faulty statement	Category
1	CarSimulator.java	28	ignition = true;	ignition = false;	CRP
2	CarSimulator.java	29	if (engine==null) {	if (engine!=null) {	ROR
3	CarSimulator.java	36	ignition = false;	ignition = true;	CRP
4	CarSimulator.java	37	engine=null;	//	SDL
5	CarSimulator.java	41	if (engine == null)	if (engine != null)	ROR
6	CarSimulator.java	45	if (throttle<(maxThrottle-5))	if (throttle>(maxThrottle-5))	ROR
7	CarSimulator.java	46	throttle +=5.0;	throttle -=5.0;	AOR
8	CarSimulator.java	56	if (throttle>0) throttle=0.0;	if (throttle<0) throttle=0.0;	ROR
9	CarSimulator.java	57	if (brakepedal<(maxBrake-1))	if (brakepedal>(maxBrake-1))	ROR
10	CarSimulator.java	58	brakepedal +=1;	brakepedal -=1;	AOR
11	CarSimulator.java	71	fspeed = fspeed+((throttle - fspeed/airResistance - 2*brakepedal))/ticksPerSecond;	fspeed = fspeed-((throttle - fspeed/airResistance - 2*brakepedal))/ticksPerSecond;	AOR
12	CarSimulator.java	72	if (fspeed>maxSpeed) fspeed=maxSpeed;	if (fspeed<maxSpeed) fspeed=maxSpeed;	ROR
13	CarSimulator.java	73	if (fspeed<0) fspeed=0;	if (fspeed>0) fspeed=0;	ROR
14	CarSimulator.java	77	if (throttle>0.0) throttle-=0.5/ticksPerSecond; //throttle decays	if (throttle>0.0) throttle+=0.5/ticksPerSecond; //throttle decays	AOR
15	CarSimulator.java	77	if (throttle>0.0) throttle-=0.5/ticksPerSecond; //throttle decays	if (throttle<0.0) throttle-=0.5/ticksPerSecond; //throttle decays	ROR
16	CarSimulator.java	91	if (throttle<0.0) throttle=0.0;	if (throttle>0.0) throttle=0.0;	ROR
17	CarSimulator.java	92	if (throttle>10.0) throttle=10.0;	if (throttle<10.0) throttle=10.0;	ROR
18	CarSimulator.java	97	return speed;	return distance;	RSR
19	SpeedControl.java	23	if (state==DISABLED) {setSpeed=0;disp.record(setSpeed);}	if (state!=DISABLED) {setSpeed=0;disp.record(setSpeed);}	ROR
20	SpeedControl.java	27	if (state==DISABLED) {	if (state!=DISABLED) {	ROR
21	SpeedControl.java	36	if (state==ENABLED) {disp.disable(); state=DISABLED;}	if (state!=ENABLED) {disp.disable(); state=DISABLED;}	ROR
22	SpeedControl.java	41	while (state==ENABLED) {	while (state!=ENABLED) {	ROR
23	SpeedControl.java	43	if (state==ENABLED) synchronized(this) {	if (state!=ENABLED) synchronized(this) {	ROR
24	SpeedControl.java	44	double error = (float)(setSpeed-cs.getSpeed())/6.0;	double error = (float)(setSpeed+cs.getSpeed())/6.0;	AOR
25	SpeedControl.java	45	double steady = (double)setSpeed/12.0;	double steady = (double)setSpeed*12.0;	AOR

Mutant #	Filename	Line #	Original statement	Faulty statement	Category
26	SpeedControl.java	46	cs.setThrottle(steady+error); //simplified feed back control	cs.setThrottle(steady-error); //simplified feed back control	AOR
27	SpeedControl.java	50	speedController=null;	//	SDL
28	Controller.java	16	if (controlState==CRUISING)	if (controlState!=CRUISING)	ROR
29	Controller.java	16	if (controlState==CRUISING)	if (controlState==ACTIVE)	CRP
30	Controller.java	16	if (controlState==CRUISING)	if (controlState==STANDBY)	CRP
31	Controller.java	17	{sc.disableControl(); controlState=STANDBY; }	{sc.disableControl(); controlState=CRUISING; }	CRP
32	Controller.java	21	if (controlState==CRUISING)	if (controlState!=CRUISING)	ROR
33	Controller.java	21	if (controlState==CRUISING)	if (controlState==ACTIVE)	CRP
34	Controller.java	21	if (controlState==CRUISING)	if (controlState==STANDBY)	CRP
35	Controller.java	22	{sc.disableControl(); controlState=STANDBY; }	{sc.disableControl(); controlState=CRUISING; }	CRP
36	Controller.java	26	if(controlState!=INACTIVE)	{if(controlState==INACTIVE) {	ROR
37	Controller.java	26	if(controlState!=INACTIVE)	{if(controlState!=ACTIVE) {	CRP
38	Controller.java	27	if (isfixed && controlState==CRUISING) sc.disableControl(); //bugfix	if (isfixed && controlState!=CRUISING) sc.disableControl(); //bugfix	ROR
39	Controller.java	27	if (isfixed && controlState==CRUISING) sc.disableControl(); //bugfix	if (isfixed && controlState==STANDBY) sc.disableControl(); //bugfix	CRP
40	Controller.java	28	controlState=INACTIVE;	controlState=ACTIVE;	CRP
41	Controller.java	33	if(controlState==INACTIVE)	if(controlState!=INACTIVE)	ROR
42	Controller.java	33	if(controlState==INACTIVE)	if(controlState==ACTIVE)	CRP
43	Controller.java	34	{sc.clearSpeed(); controlState=ACTIVE;}	{sc.clearSpeed(); controlState=CRUISING;}	CRP
44	Controller.java	38	if(controlState!=INACTIVE){	if(controlState==INACTIVE){	ROR
45	Controller.java	38	if(controlState!=INACTIVE){	if(controlState!=ACTIVE){	CRP
46	Controller.java	40	controlState=CRUISING;	controlState=STANDBY;	CRP
47	Controller.java	45	if(controlState==CRUISING)	if(controlState!=CRUISING)	ROR
48	Controller.java	45	if(controlState==CRUISING)	if(controlState==ACTIVE)	CRP
49	Controller.java	45	if(controlState==CRUISING)	if(controlState==STANDBY)	CRP
50	Controller.java	46	{sc.disableControl(); controlState=STANDBY;}	{sc.disableControl(); controlState=CRUISING;}	CRP
51	Controller.java	50	if(controlState==STANDBY)	if(controlState!=STANDBY)	ROR
52	Controller.java	50	if(controlState==STANDBY)	if(controlState==CRUISING)	CRP
53	Controller.java	51	{sc.enableControl(); controlState=CRUISING;}	{sc.enableControl(); controlState=STANDBY;}	CRP
54	Controller.java	17	{sc.disableControl(); controlState=STANDBY; }	{sc.enableControl(); controlState=STANDBY; }	MNR
55	Controller.java	22	{sc.disableControl(); controlState=STANDBY; }	{sc.enableControl(); controlState=STANDBY; }	MNR
56	Controller.java	27	if (isfixed && controlState==CRUISING) sc.disableControl(); //bugfix	if (isfixed && controlState==CRUISING) sc.enableControl(); //bugfix	MNR
57	Controller.java	34	{sc.clearSpeed(); controlState=ACTIVE;}	{sc.recordSpeed(); controlState=ACTIVE;}	MNR
58	Controller.java	39	sc.recordSpeed(); sc.enableControl();	sc.recordSpeed(); sc.disableControl();	MNR
59	Controller.java	39	sc.recordSpeed(); sc.enableControl();	sc.clearSpeed(); sc.enableControl();	MNR

Mutant #	Filename	Line #	Original statement	Faulty statement	Category
60	Controller.java	46	{sc.disableControl(); controlState=STANDBY;}	{sc.enableControl(); controlState=STANDBY;}	MNR
61	Controller.java	51	{sc.enableControl(); controlState=CRUISING;}	{sc.disableControl(); controlState=CRUISING;}	MNR
62	CarSimulator.java	42	throttle = 0;	throttle = 1;	CRP
63	CarSimulator.java	44	if (brakepedal>0) brakepedal=0;	if (brakepedal<0) brakepedal=0;	ROR
64	CarSimulator.java	44	if (brakepedal>0) brakepedal=0;	if (brakepedal>0) brakepedal=1;	CRP
65	CarSimulator.java	48	throttle=maxThrottle;	throttle=maxThrottle+1;	CRP
66	CarSimulator.java	53	if (engine == null)	if (engine != null)	ROR
67	CarSimulator.java	54	brakepedal = 0;	brakepedal = 1;	CRP
68	CarSimulator.java	60	brakepedal=maxBrake;	brakepedal=maxBrake+1;	CRP
69	CarSimulator.java	81	speed=0; //no freewheeling!!	//	SDL
70	CarSimulator.java	83	throttle=0;	//	SDL
71	CarSimulator.java	84	brakepedal=0;	//	SDL
72	Controller.java	8	private int controlState = INACTIVE; //initial state	private int controlState = ACTIVE; //initial state	CRP
73	Controller.java	17	{sc.disableControl(); controlState=STANDBY; }	{sc.recordSpeed(); controlState=STANDBY; }	MNR
74	Controller.java	17	{sc.disableControl(); controlState=STANDBY; }	{sc.clearSpeed(); controlState=STANDBY; }	MNR
75	Controller.java	22	{sc.disableControl(); controlState=STANDBY; }	{sc.recordSpeed(); controlState=STANDBY; }	MNR
76	Controller.java	22	{sc.disableControl(); controlState=STANDBY; }	{sc.clearSpeed(); controlState=STANDBY; }	MNR
77	Controller.java	27	if (isfixed && controlState==CRUISING) sc.disableControl(); //bugfix	if (isfixed && controlState==CRUISING) sc.recordSpeed(); //bugfix	MNR
78	Controller.java	27	if (isfixed && controlState==CRUISING) sc.disableControl(); //bugfix	if (isfixed && controlState==CRUISING) sc.clearSpeed(); //bugfix	MNR
79	Controller.java	34	{sc.clearSpeed(); controlState=ACTIVE;}	{sc.enableControl(); controlState=ACTIVE;}	MNR
80	Controller.java	34	{sc.clearSpeed(); controlState=ACTIVE;}	{sc.disableControl(); controlState=ACTIVE;}	MNR
81	Controller.java	46	{sc.disableControl(); controlState=STANDBY;}	{sc.recordSpeed(); controlState=STANDBY;}	MNR
82	Controller.java	46	{sc.disableControl(); controlState=STANDBY;}	{sc.clearSpeed(); controlState=STANDBY;}	MNR
83	Controller.java	51	{sc.enableControl(); controlState=CRUISING;}	{sc.recordSpeed(); controlState=CRUISING;}	MNR
84	Controller.java	51	{sc.enableControl(); controlState=CRUISING;}	{sc.clearSpeed(); controlState=CRUISING;}	MNR
85	Controller.java	17	{sc.disableControl(); controlState=STANDBY; }	sc.disableControl();	SDL
86	Controller.java	22	{sc.disableControl(); controlState=STANDBY; }	sc.disableControl();	SDL
87	Controller.java	28	controlState=INACTIVE;	//	SDL
88	Controller.java	34	{sc.clearSpeed(); controlState=ACTIVE;}	sc.clearSpeed();	SDL
89	Controller.java	40	controlState=CRUISING;	//	SDL

Mutant #	Filename	Line #	Original statement	Faulty statement	Category
90	Controller.java	46	{sc.disableControl(); controlState=STANDBY;}	sc.disableControl();	SDL
91	Controller.java	51	{sc.enableControl(); controlState=CRUISING;}	sc.enableControl();	SDL

Table 8. Mutant list for Cruise Control case study

Appendix F Mutants Designed for OrdSet Case Study

This appendix provides the detailed information about the faults seeded for OrdSet case study.

Mutant#	Filename	Line#	Original statement	Faulty statement	Category
1	Ordset.cpp	71	if (_last < _set_size)	int _last; if (_last < _set_size)	SCO
2	Ordset.cpp	71	if (_last < _set_size)	if (_last <= _set_size)	LOR
3	Ordset.cpp	71	if (_last < _set_size)	if (_last < _set_size - 1)	LCO
4	Ordset.cpp	72	return _last++;	return _set_size++;	VRO
5	Ordset.cpp	72	return _last++;	return last;	LOR
6	Ordset.cpp	72	return _last++;	return _last--;	LOR
7	Ordset.cpp	72	return _last++;	return _resized_times;	VRO
8	Ordset.cpp	80	if (n <= min_set_size) return min_set_size;	if (n <= min_set_size - 1) return min_set_size;	LCO
9	Ordset.cpp	80	if (n <= min_set_size) return min_set_size;	if (n > min_set_size) return min_set_size;	LOR
10	Ordset.cpp	80	if (n <= min_set_size) return min_set_size;	if (n <= min_set_size) return min_set_size + 1;	LCO
11	Ordset.cpp	80	if (n <= min_set_size) return min_set_size;	if (n <= max_accepted_resizes) return min_set_size;	SRC
12	Ordset.cpp	80	if (n <= min_set_size) return min_set_size;		SDL
13	Ordset.cpp	84	return (mod+1)*min_set_size;	return mod*min_set_size;	LCO
14	Ordset.cpp	84	return (mod+1)*min_set_size;	return (mod+1)*max_accepted_resizes;	SRC
15	Ordset.cpp	107	int new_size=_set_size+min_set_size;	int new_size=_set_size-min_set_size;	LOR
16	Ordset.cpp	107	int new_size=_set_size+min_set_size;	int new_size=_set_size+max_accepted_resizes;	SRC
17	Ordset.cpp	107	int new_size=_set_size+min_set_size;	int new_size=_last+min_set_size;	VRO
18	Ordset.cpp	109	if (new_size < max_set_size &&	if (new_size < max_set_size	LOR
19	Ordset.cpp	110	_resized_times < max_accepted_resizes){	_set_size < max_accepted_resizes){	VRO
20	Ordset.cpp	110	_resized_times < max_accepted_resizes){	_resized_times <= max_accepted_resizes){	LOR
21	Ordset.cpp	110	_resized_times < max_accepted_resizes){	_resized_times < min_set_size){	SRC
22	Ordset.cpp	112	int *_new_set=new int[new_size];	int *_new_set=new int[_set_size];	VRO
23	Ordset.cpp	115	for (k = 0; k < _last+1; k++)	for (k = 0; k < _last; k++)	LCO
24	Ordset.cpp	116	_new_set[k] = _set[k];	_new_set[k] = _set[k+1];	LCO
25	Ordset.cpp	117		return;	CFD

Mutant#	Filename	Line#	Original statement	Faulty statement	Category
26	Ordset.cpp	118	_set_size=new_size;	_set_size=_last;	VRO
27	Ordset.cpp	118	_set_size=new_size;	_set_size=new_size + 1;	LCO
28	Ordset.cpp	118	_set_size=new_size;		SDL
29	Ordset.cpp	122	_resized_times++;		SDL
30	Ordset.cpp	124	_resize_error ++;		SDL
31	Ordset.cpp	133	_set_size=defSetSize(size);	int _set_size; _set_size=defSetSize(size);	SCO
32	Ordset.cpp	133	_set_size=defSetSize(size);	_set_size=make_a_free_slot(size);	MNR
33	Ordset.cpp	134	_set= new int[_set_size];	_set= new int[_last];	VRO
34	Ordset.cpp	135	_last=- 1;	int _last; _last=-1;	SCO
35	Ordset.cpp	135	_last=- 1;	_last=0;	LCO
36	Ordset.cpp	136	_resized_times = 0;	_resized_times = -1;	LCO
37	Ordset.cpp	137	_resize_error = 0 ;	_resize_error = -1;	LCO
38	Ordset.cpp	176	return binSearch(_set, _last+1,n)>=0 ;	return binSearch(_set, n, _last+1)>=0 ;	AOC
39	Ordset.cpp	176	return binSearch(_set, _last+1,n)>=0 ;	return binSearch(_set, _last+1,n)>0 ;	LOR
40	Ordset.cpp	176	return binSearch(_set, _last+1,n)>=0 ;	return binSearch(_set, _last+1,n)<0 ;	LOR
41	Ordset.cpp	176	return binSearch(_set, _last+1,n)>=0 ;	return binSearch(_set, _last,n)>=0 ;	LCO
42	Ordset.cpp	232	int where = binSearch(_set, _last + 1, val);	int where = binSearch(_set, val, _last + 1);	AOC
43	Ordset.cpp	232	int where = binSearch(_set, _last + 1, val);	int where = binSearch(_set, _last, val);	LCO
44	Ordset.cpp	232	int where = binSearch(_set, _last + 1, val);	int where = binSearch(_set, _set_size + 1, val);	VRO
45	Ordset.cpp	234	if (where >=0){	if (where <0){	SSO
46	Ordset.cpp	237	for (k=where; k< _last; k++)	for (k=0; k< _last; _k++)	VRO
47	Ordset.cpp	237	for (k=where; k< _last; k++)	for (k=where; k< _last; _last++)	VRO
48	Ordset.cpp	237	for (k=where; k< _last; k++)	for (k=where; k< _last - 1 ; k++)	LCO
49	Ordset.cpp	237	swap if and _last--		SSO
50	Ordset.cpp	238	_set[k] = _set[k+1];	_set[k] = _set[k];	SDL
51	Ordset.cpp	240	_last --;	_last ++;	LOR
52	Ordset.cpp	240	_last --;		SDL
53	Ordset.cpp	254	if (*this == n) return *this;		SDL
54	Ordset.cpp	254	swap two ifs		SSO
55	Ordset.cpp	256	if (_last+1>=_set_size)	if (_last>=_set_size)	LCO
56	Ordset.cpp	256	if (_last+1>=_set_size)	if (_last+1>_set_size)	LOR
57	Ordset.cpp	256	if (_last+1>=_set_size)	if (_last+1<_set_size)	LOR
58	Ordset.cpp	257	resizeArray();	{resizeArray(); return *this; }	CFD
59	Ordset.cpp	259	_set[make_a_free_slot(n)]=n;	_set[_last]=n;	VRO
60	Ordset.cpp	259	_set[make_a_free_slot(n)]=n;	_set[defSetSize(n)]=n;	MNR
61	Ordset.cpp	260	updateLast();		SDL
62	Ordset.cpp	260	updateLast();	getSetLast();	MNR
63	Ordset.cpp	259	swap 259 and 260		SSO
64	Ordset.cpp	281	register int where=_last+1;	register int where=_last;	LCO
65	Ordset.cpp	284	if (where>=_set_size){	if (where>=_last){	VRO

Mutant#	Filename	Line#	Original statement	Faulty statement	Category
66	Ordset.cpp	284	if (where>=_set_size){	if (where>_set_size){	LOR
67	Ordset.cpp	285	if (new_size < max_set_size &&	if (new_size < max_set_size	LOR
68	Ordset.cpp	285	if/else swapped (SSO)		SSO
69	Ordset.cpp	286	_resized_times < max_accepted_resizes)	_resized_times <= max_accepted_resizes)	LOR
70	Ordset.cpp	286	_resized_times < max_accepted_resizes)	_resized_times > max_accepted_resizes)	LOR
71	Ordset.cpp	286	_resized_times < max_accepted_resizes)	_resized_times < min_set_size)	SRC
72	Ordset.cpp	287	resizeArray();	{resizeArray(); return where;}	CFD
73	Ordset.cpp	294	while((where-1>=0) && (_set[where-1] > n)){	while((where-1>=0) (_set[where-1] > n)){	LOR
74	Ordset.cpp	294	while((where-1>=0) && (_set[where-1] > n)){	while((where>=0) && (_set[where-1] > n)){	LCO
75	Ordset.cpp	294	while((where-1>=0) && (_set[where-1] > n)){	while((where-1>=0) && (_set[where] > n)){	LCO
76	Ordset.cpp	294	while((where-1>=0) && (_set[where-1] > n)){	while((where-1>=0) && (_set[where-1] < n)){	LOR
77	Ordset.cpp	295	_set[where]=_set[where-1];	_set[where]=_set[where];	SDL
78	Ordset.cpp	295	swapped where-- and assignment		SSO
79	Ordset.cpp	295	_set[where]=_set[where-1];	_set[where]=_set[where-1]; break;	CFD
80	Ordset.cpp	296	where--;		SDL
81	Ordset.cpp	296	where--;	where--; break;	CFD
82	Ordset.cpp	299	return where;	return _last;	VRO
83	Ordset.cpp	482	int binSearch(int *a, int size,int x)	int binSearch(int *a, int x,int size)	POC
84	Ordset.cpp	485	register long j=size-1;	register long j=size;	LCO
85	Ordset.cpp	488	if (j<0) return -1;	if (j<0) return 0;	LCO
86	Ordset.cpp	488	if (j<0) return -1;	if (j<=0) return -1;	LOR
87	Ordset.cpp	488	if (j<0) return -1;		SDL
88	Ordset.cpp	492	if (a[j] < x) return -1;	if (a[j] <= x) return -1;	LOR
89	Ordset.cpp	492	if (a[j] < x) return -1;	if (a[j] < x) return 0;	LCO
90	Ordset.cpp	492	if (a[j] < x) return -1;		SDL
91	Ordset.cpp	494	while(i<j){	while(i>j){	LOR
92	Ordset.cpp	497	if (x>a[m])	if (x<=a[m])	SSO
93	Ordset.cpp	498	i=m+1;	i=m;	LCO
94	Ordset.cpp	498	i=m+1;	i=j+1;	VRO
95	Ordset.cpp	498	i=m+1;	i=i+1;	VRO
96	Ordset.cpp	500	j=m;	j=i;	VRO
97	Ordset.cpp	500	j=m;	j=m+1;	LCO
98	Ordset.cpp	504	if (x == a[i]) return i;	if (x != a[i]) return i;	LOR
99	Ordset.cpp	504	if (x == a[i]) return i;	if (x == a[i]) return -1;	VRO
100	Ordset.cpp	504	if (x == a[i]) return i;	if (x == a[i]) return i-1;	LCO
101	Ordset.cpp	506	return -1;	return 0;	LCO

Table 9. Mutant list for OrdSet case study

Appendix G Analysis of Live Mutants with Transition Trees (Cruise Control)

This appendix gives the detailed analysis about the nonequivalent mutants missed by the three transition trees (Figure 4) in the first case study of Cruise Control. The missing mutants are listed in Table 10.

Transition Tree	Live Mutant #
Transition Tree 1	8, 16, 24, 25, 26, 59, 69
Transition Tree 2	63, 64, 71
Transition Tree 3	4, 8, 16, 22, 24, 25, 26, 59, 63, 64, 69, 71

Table 10. Live Mutants with Transition Trees (Cruise Control)

Mutant #4 was seeded according to the mutation operator Statement Deletion (SDL) by comment out the statement `engine=null` in the `engineOff` method in class `CarSimulator`. The purpose of this statement is to kill the thread in class `CarSimulator` when the engine is turned off. When the thread is killed, the `run` method in class `CarSimulator` will set the value of speed and brake pedal back to zero. If the event accelerator or brake were not executed before `engineOff`, the value of speed and brake pedal would always be zero, so that the mutant provides the same outputs as the original statement does. Therefore, this fault can be only detected by the test sequence that has event accelerator or brake before event `engineOff`.

Mutant #8 was seeded according to the mutation operator Relational Operator Replacement (ROR) by replacing the statement of `if (throttle>0) throttle = 0.0;` with `if (throttle<0) throttle = 0.0;` in the `brake` method in class

CarSimulator. The original statement sets the throttle to zero if it was greater than zero when brake is called while the mutant does the same when throttle is less than zero. Only event accelerator can initially change the value of throttle. If there were no accelerator event before brake, the value of throttle would always be zero, so that this statement would not be actually executed. Therefore, this fault can only be detected by the test sequence that contains event accelerator before event brake.

Mutant #16 was seeded according to the mutation operator ROR by replacing the statement `if (throttle<0.0) throttle=0.0;` with `if (throttle>0.0) throttle=0.0;` in the `setThrottle` method in class `CarSimulator`. The `setThrottle` method is called when cruise control is turned on. Only event accelerator can initially change the value of throttle to a number other than zero. If there were no accelerator event before on, the value of throttle would always remain in zero, so the original statement and the mutant would lead to the same throttle value, i.e., zero. Therefore, this fault can only be detected by the test sequence that has event accelerator before event on.

Mutant #22 was seeded according to the mutation operator ROR by replacing the statement `while (state==ENABLED){` with `while (state!=ENABLED){` in the `run` method in class `SpeedControl`. This mutant changes the condition for entering the `run` method that implements the speed control function when cruise control is turned on or resumed. If the accelerator or brake event were not called to change the value of speed and brake pedal before event on or resume, no matter whether this function was executed correctly or not, the value of car speed and brake pedal would always stay in zero, so that the original code and the mutant lead to the same

outputs. Therefore, this fault can only be detected by the test sequence that has event accelerator or brake before event on or resume.

Mutant #24 was seeded according to the mutation operator Arithmetic Operator Replacement (AOR) by replacing the statement of `double error=(float)(setSpeed-cs.getSpeed())/6.0;` with `double error=(float)(setSpeed+cs.getSpeed())/6.0;` in the run method that implement the speed control function in class `SpeedControl` when cruise control is turned on. When cruise control is turned on, the car speed will be maintained at a certain value. This fault can only be detected by the test sequence that has event accelerator before event on, which means the speed has to be changed to a value other than zero by calling accelerator first. Otherwise, the speed will remain in zero, so the mutant gives the same outputs as the original code does.

Mutant #25 was seeded according to the mutation operator AOR by replacing the statement of `double steady=(double)setSpeed/12.0;` with `double steady=(double)setSpeed*12.0;` in the run method in class `SpeedControl`. For the same reason explained for mutant #24, this fault can only be detected by the test sequence that has event accelerator before event on.

Mutant #26 was seeded according to the mutation operator AOR by replacing the statement of `cs.setThrottle(steady+error);` with `cs.setThrottle(steady-error);` in the run method in class `SpeedControl`. For the same reason explained for mutant #24, this fault can only be detected by the test sequence that has event accelerator before event on.

Mutant #59 was seeded according to the mutation operator Method Name Replacement (MNR) by replacing the statement of `sc.recordSpeed();`
`sc.enableControl();` with `sc.clearSpeed();` `sc.enableControl();`
in the `on` method in class `Controller`. The `recordSpeed` method records the current speed while the `clearSpeed` method sets the speed to zero. If there were no accelerator event was executed, the value of the current speed would always be zero, so that the original code and the mutant lead to the same outputs. Therefore, this fault can only be detected by the test sequence that has event accelerator before event `on`.

Mutant #63 was seeded according to the mutation operator ROR by replacing the statement `if (brakepedal>0) brakepedal=0;` with `if (brakepedal<0) brakepedal=0;` in the `accelerator` method in class `CarSimulator`. If the brake pedal never were changed to a value other than zero before `accelerator` was called, the value of `brakepedal` would be zero, so that this statement would not be actually executed. The original code and the mutant lead to the same outputs. Therefore, this fault can only be detected by the test sequence that has event `brake` before event `accelerator`.

Mutant #64 was seeded according to the mutation operator Constant Replacement (CRP) by replacing the statement `if (brakepedal>0) brakepedal=0;` with `if (brakepedal>0) brakepedal=1;` in the `accelerator` method in class `CarSimulator`. For the same reason explained for mutant #63, this fault can only be detected by the test sequence that has event `brake` before event `accelerator`.

Mutant #69 was seeded according to the mutation operator Statement Deletion (SDL) by comment out the statement of `speed=0; //no freewheeling!!` in the `run` method in class `CarSimulator`. It sets the speed back to zero when the thread in class `CarSimulator` is killed. This fault can only be detected by the test sequence that has event `accelerator` before event `engineOff` which kills the thread. Otherwise, the speed would always remain in zero, no matter whether this statement exists or not, so that the mutant would give the same outputs as the original code does.

Mutant #71 was seeded according to the mutation operator SDL by comment out the statement `brakepedal=0` in the `run` method in class `CarSimulator`. It sets the brake pedal back to zero when `engineOff` is called to kill the thread in class `CarSimulator`. If the brake event is not called, the value of brake pedal always equals to zero, no matter if this statement exists or not. So that the mutant would provide the same outputs as the original code does. Therefore, this fault can be only detected by the test sequence that has event `brake` before event `engineOff`.

Appendix H Analysis of Live Mutants with ATP, TT and FP Adequate Test Sets (OrdSet)

This appendix gives the detailed analysis about the nonequivalent mutants missed by ATP, TT and FP adequate test sets in the second case study of OrdSet. The missing mutants are listed in Table 11.

Criterion	Live Mutant #
ATP	8, 9, 11, 12, 47, 48, 49, 50, 81, 96
TT	8, 9, 11, 12, 46, 47, 48, 49, 50, 81, 91, 94, 96
FP	8, 9, 11, 12, 96

Table 11. Live Mutants with ATP, TT and FP Adequate Test Sets (OrdSet)

Mutant #8, #9, #11 and #12 were created by seeding faults in method `defSetSize(int n)`. This method defines the initial size of the ordered set according to the actual parameter of constructor `OrdSet(int n)`. The size of the set is defined as 3 when the actual parameter n is less or equal to 3, $(n/3 + 1) * 3$ otherwise. It can be seen that when n equals to 1 and 2, the size is always defined as 3. These four faults can only be revealed when the set is instantiated by `OrdSet(3)`. Five ATP adequate test sets, thirteen TT adequate test sets and eleven FP test sets missed these four mutants.

Mutant #46 was seeded by replacing statement `for(k=where; k<_last; k++)` with statement `for(k=0; k<_last; k++)` in method `remove(int n)`. The value of variable `where` holds the position of n which is the element need to be removed from the ordered set. The value of 0 indicates the first element (the smallest

integer) in the ordered set. In order to differentiate the mutant and the original statement, where has to have a value other than 0, i.e. the element need to be removed is not the smallest one in the ordered set. As it happens, the actual parameter of method remove in sixteen TT adequate test sets is the first element in the ordered set. Therefore, this mutant was missed by these test sets.

Mutant #47, #48, #49 and #50 were created by seeding faults in method `remove(int n)`. When an element is removed from the ordered set, the positions of the remaining elements are adjusted, i.e. the blank left by removing `n` is filled up. All these four mutants cause the incorrect arrangement for the elements position. The faults can be only detected when `_last >= where + 1` (`_last` is the index of the last element and `where` is the position of the element need to be removed), which means `n` has to be in the position at least two elements ahead of the last element. For example, suppose there are three elements in the set, then if and only if removing the first (smallest) element will kill these mutants. One ATP adequate test set and twelve TT adequate test sets missed these four mutants.

Mutant #81 was seeded by adding a break statement in method `make_a_free_slot(int n)` to change the control flow disruption when rearranging the order of the elements according to the value of the new added element when `operator +=` is invoked. `make_a_free_slot` returns the position the integer `n` will be added in. The fault can only be detected when two conditions are satisfied. First, both `where - 1 >= 0` and `(where - 1) - 1 >= 0` (`where` equals to `_last + 1`) have to be true, which implies that `_last` has to be greater than 1, i.e., the set has to contain at least two elements. Second, both `_set[where - 1] > n` and `_set[(where - 1) -`

$1] > n$ have to be true, which implies that the new added element has to be in the position at least two elements ahead of the last element. For example, if there are two elements in the set, if and only if adding an integer smaller than the existing elements will detect the fault. Three ATP adequate test sets and twenty-three TT adequate test sets missed this mutant.

Mutant #91 and #94 were created by seeding faults in method `binSearch` that implements the binary search algorithm. Method `binSearch` returns the position of a given integer if it is in the set or returns `-1` otherwise. This method is called when `add/remove` an integer into/from the set. These two mutants wrongly implement the binary search. #91 changes the searching loop condition whereas #94 makes the update of the index incorrect. The return values with the mutants are different from the original code only when the set contains at least two elements when event `operator+=1` and `remove1`²⁰ are invoked and then the integer need to be added/removed is not the first element in the set. Fourteen TT adequate test sets missed these two mutants.

Mutant #96 was also seeded in method `binSearch` as for mutant #91 and #94. It results an incorrect divide procedure in binary search. The faults can only be detected if the set contains at least three elements when event `operator+=1` and `remove1` are invoked. Furthermore, the integer need to be added/removed into/from the set has to be in the intermediary position in the ordered set. For example, if there are three elements in the set, only removing element in the middle position, or trying adding a duplicate integer that is already in the set and is between the smallest and greatest element in the set can

²⁰ Event `operator+=1` and `remove1` are equivalent with `operator+=(int n)` and `remove(int n)` with guard condition `[self->includes(n)]`, respectively.

kill this mutant. Three ATP adequate test sets, twenty-one TT adequate test sets and one FP adequate test set missed this mutant.

Appendix I System Inputs and Outputs for IFTR

A set of input and output files are defined based on the requirements from the experiment design. This appendix provides details about the system inputs and outputs for IFTR.

1. System Inputs

Statechart model file. This file defines the statechart specifications and may have different formats depending on the notation used for representing the model of the SUT. For example, if the file takes the format of a transition table, then each transition is represented by a tuple (the source state, the trigger events, the guard condition, action, resultant state).

Infeasible paths file. This file stores the list of infeasible paths if there are any in the statechart.

Criteria list file. This file stores the list of criteria that have been implemented in the system. The criteria names in this file have to be unique, and will be used as the criterion identification in the implementation. The user can add new criteria as needed.

Mutant list file. This file lists the faults which will be seeded. In the mutant list file, the user should define the mutant number, the name of the class that the fault will be seeded in, the line number of the statement that will be replaced, the original statement, the mutant statement and the mutation operator category that the fault belongs to.

Test oracle file. This file is created by the user-defined test driver that runs the test cases on the original version of the SUT. It contains all the expected outputs for each test case, i.e., the expected value of state invariant in the resultant state and the values of the variables in the response.

Actual output file. This file is created by the user-defined test driver that runs test cases on mutants. This file stores the actual values of the variables used in the resultant state checking and the response checking when the test driver executes test cases on a given mutant. The actual output will be compared with the expected outputs to see if the mutant causes a failure. This will also be helpful when the researcher analyzes why certain mutants are not detected by the test cases.

Test result file. This file is created by the user-defined test driver. This file stores the test result (passed or fail) after executing test cases on a given mutant. Each line is an integer of either 0 or 1, in which 1 indicates the mutant is killed by the corresponding test case, whereas 0 indicates the mutant is not killed.

Equivalent mutant list file. This file stores the list of equivalent mutant numbers. There may exist certain mutants which are not detected by any test case in the test pool. IFTR will provide the list of potential equivalent mutants to the user. The user has to identify whether a live mutant is equivalent, and then record the list of all equivalent mutants in the mutant set. The equivalent mutants are not physically removed from the mutant set, but are not counted when calculating mutation scores.

2. Intermediate Results

Criterion feature files. These files store the test requirements, i.e., the features for selected criteria in the experiment.

Test pool file. This file stores a large number of distinct test cases. Each test case is a transition sequence, starting from the initial state.

Coverage matrix files. Each criterion has its own coverage matrix. The rows are the test cases and the columns are the features of the given criterion. An entry of “1” in position (i, j) indicates that test case i covers feature j . A line that summarizes the coverage of each feature is given at the bottom of the file.

Result matrix file. The rows in the result matrix are the test cases and the columns are the mutants. An entry of “1” in position (i, j) indicates that test case i kills the mutant j . The order of the rows in the result matrix is identical to the one in the coverage matrices.

3. System Outputs

Simulation result files. These files are created as the simulation result for each selected criterion. There are two files for each selected test adequacy criterion, one for the test criterion and another for the Null criterion with the comparable test set size. These files contain the information about the test sets, such as test set size, coverage ratio and mutation score.

Appendix J Use Case Description for IFTR

This appendix presents the detailed description of the use cases shows in Figure 21 for IFTR in terms of the use case name, the entry condition, the flow of events and the exit condition²¹. Researcher is the actor in the system.

Use case name	Load Model
Entry condition	<ol style="list-style-type: none">1. The statechart model file has been created by Researcher.2. Researcher has recorded the infeasible paths in the infeasible paths file.3. Researcher selects Load Model.
Flow of events	<ol style="list-style-type: none">4. Researcher indicates the filename for the statechart model file.5. System loads the statechart model file.6. System parses the model and obtains the objects of the states and the transitions.
Exit condition	<ol style="list-style-type: none">7. The information of the statechart has been obtained.

Use case name	Generate Mutants
Entry condition	<ol style="list-style-type: none">1. Researcher has created the mutant list file.2. Researcher selects Generate Mutants.
Flow of events	<ol style="list-style-type: none">3. Researcher specifies the location of the SUT and the mutant list file.4. System loads the original SUT.5. System reads the mutant list file.6. System creates mutants one by one, according to the mutant list file.
Exit condition	<ol style="list-style-type: none">7. The generation of the mutants is complete.

²¹ Refer to Appendix I for a detailed explanation of the input and output files.

Use case name	Analyze Features
Entry condition	1. The given model has been loaded into the system. 2. Researcher selects Analyze Features.
Flow of events	3. Researcher specifies the test criteria. 4. System generates the test sequences according to the requirements of the given criteria. 5. System creates the criterion feature file for each selected criterion.
Exit condition	6. The corresponding feature files have been created.

Use case name	Construct Test Pool
Entry condition	1. The statechart model has been loaded into the system. 2. Researcher selects Construct Test Pool.
Flow of events	3. System reads in the statechart model. 4. System randomly traverses the statechart diagram to generate test cases, i.e., transition sequences that start from the initial state, and store the sequences in a temp file.
Exit condition	5. A large number of test cases has been generated and stored in the temp file.

Use case name	Prune Test Pool
Entry condition	1. The test pool temp file has been created. 2. Researcher selects Prune Test Pool.
Flow of events	3. System identifies and eliminates the identical test cases in the temp file. The distinct test cases are recorded in the test pool file.
Exit condition	4. Identical test cases have been removed and the test pool file has been created.

Use case name	Build Coverage Matrices
Entry condition	1. The criterion feature files have been created. 2. The test pool file has been created. 3. Researcher selects Build Coverage Matrix.

Flow of events	4. System creates a coverage matrix for each selected criterion and records the matrix into a file.
Exit condition	5. The coverage matrix file(s) is (is) created.

Use case name Add Test Cases

Entry condition	1. The features for all selected test criteria are created.
2. Researcher selects Add Test Cases.
Flow of events	3. System adds new test cases, either from a text file or by converting criterion features to test cases that start from the initial state, as per the researcher's choice.
4. System appends the additional test cases to the test pool temp file.
Exit condition	5. The test pool is adequate for all selected criteria.

Use case name Create Oracle

Entry condition	1. The modification of the test pool has been finalized.
2. The executable test driver for the given SUT has been developed by Researcher.
3. Researcher selects Create Oracle.
Flow of events	4. System executes the test driver provided by the researcher that runs all test cases in the test pool on the original version of the SUT.
5. The expected values of the state variables and the response are recorded in the test oracle file.
Exit condition	6. The test oracle file is created.

Use case name Execute Test Cases

Entry condition	1. The mutant set has been created.
2. The modification of the test pool has been finalized.
3. The test oracle file has been created.
4. The executable test driver for the given SUT has been developed by Researcher.
5. Researcher selects Run Test. |
|------------------------|--|

Flow of events	6. System executes the user-defined test driver that runs all test cases one by one on the mutants. 7. The test driver records the given mutant is killed or not killed by which test cases in the test result file for each mutant.
-----------------------	---

Exit condition	8. The execution of the test cases is complete.
-----------------------	---

Use case name	Build Result Matrix
----------------------	----------------------------

Entry condition	1. The execution of the test cases is completed. 2. Researcher selects Build Result Matrix.
------------------------	--

Flow of events	3. System generates the result matrix according to the test results and records it in the result matrix file.
-----------------------	---

Exit condition	4. The result matrix file is created.
-----------------------	---------------------------------------

Use case name	Identify Equivalent Mutants
----------------------	------------------------------------

Entry condition	1. There exist mutants that are not killed by any test case in the test pool.
------------------------	---

	2. Researcher selects Identify Equivalent Mutants.
--	--

Flow of events	3. System provides a list of the mutants remaining alive. 4. Researcher determines whether these mutants are equivalent mutant and records the list of equivalent mutants in the equivalent mutant list file.
-----------------------	--

Exit condition	5. All equivalent mutants are identified and the equivalent mutant list file is created.
-----------------------	--

Use case name	Perform Simulation
----------------------	---------------------------

Entry condition	1. The coverage matrix files have been created. 2. The result matrix file has been created. 3. Researcher selects Perform Simulation.
------------------------	---

Flow of events	4. System generates 100 C-adequate test sets based on the coverage matrices and the result matrix for each test adequacy criterion, according to the test set generation criterion defined in the experiment design. 5. System generates 100 test sets with the equivalent size corresponding to each adequacy criterion for the Null criterion, by randomly selecting test cases from the test pool.
-----------------------	--

	6. System records the test set size, coverage ratio and mutation score of the test sets in files.
Exit condition	7. The simulation result files are created.
Use case name	Analyze Data
Entry condition	1. The simulation result files have been created. 2. Researcher selects a statistical analysis software package.
Flow of events	3. Researcher loads the simulation result files into the chosen statistical package. 4. Researcher applies the appropriate statistic analysis technologies to determine the model of the relationships under investigation. 5. The package plots the data in diagrams and outputs the statistics about the data.
Exit condition	6. The desired statistics and diagrams are obtained.