On the Round Trip Path Strategy for State Based Testing

By

May Farouk Khalil

A thesis submitted to
The Faculty of Graduate Studies and Research
In partial fulfilment of
The degree requirements of
Master of Applied Science

Ottawa-Carleton Institute for
Electrical and Computer Engineering (OCIECE)

Department Electrical and Computer Engineering
Carleton University
Ottawa, Ontario, Canada
September 2010

Copyright © 2010 - May Khalil

The undersigned recommends to
NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

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

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

AVIS:

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l’Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

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

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n’y aura aucun contenu manquant.
Abstract

A number of techniques have been proposed for state-based class testing. Some of these techniques are based on deriving test sequences using transition trees generated by traversing state machine diagrams. Different traversal algorithms can be used to generate the transition trees, which then vary in their resulting testing cost and fault detection effectiveness. Several hypotheses are made when one derives test cases using transitions trees. Among others, it is assumed that covering paths in the tree is equivalent to covering round-trip paths, i.e., path that start at a state and end at the same state without other loops, and it is assumed that different traversal algorithms are equivalent. In this research we investigate whether these assumptions hold in practice. We attempt to generate all possible transition trees using two different well-known graph traversal algorithms—the depth first and the breadth first algorithms, and we analyse the main characteristics of the generated trees and their paths and compare them to covering round trip paths.

Given the results of this investigation, this research also proposes a new algorithm to generate a test suite from a state model and evaluates its effectiveness in comparison to previous algorithms. Finally recommendations for best practice techniques to generate test suites from state machine diagrams are presented.
Acknowledgment

I would like to sincerely thank my supervisor Dr. Labiche for his continued support, patient guidance, and help throughout my thesis work.

I am grateful to my colleagues in SQUALL Lab who provided invaluable assistance and motivations.

Warmest thanks to my family for the encouragement and moral support they have given me. Without them, I would not have been able to achieve this.
Table of Contents

Table of Contents .................................................................................................................. 5
List of Tables .......................................................................................................................... 8
List of Figures .......................................................................................................................... 10
Chapter 1 : Introduction ........................................................................................................ 13
  1.1 Background .................................................................................................................... 13
  1.2 Main contributions .......................................................................................................... 17
  1.3 Thesis organization ......................................................................................................... 17
Chapter 2 : Related Work ...................................................................................................... 18
  2.1 State-Based Criteria other than Round trip path coverage ........................................ 19
  2.2 Round trip path coverage .............................................................................................. 19
  2.3 Previous experiments with state based testing ............................................................... 26
Chapter 3 : Experimental Design .......................................................................................... 30
  3.1 Case Study Systems ....................................................................................................... 32
    3.1.1 CruiseControl Case Study ..................................................................................... 33
    3.1.2 OrderedSet Case Study System ............................................................................ 33
  3.2 Test Suite Generation ..................................................................................................... 37
    3.2.1 Algorithm for generating transition trees ............................................................... 38
    3.2.2 Test Cases Covering Round Trip Paths Fully ...................................................... 39
    3.2.3 Classification of Test Cases .................................................................................. 41
  3.3 Mutant Generation ......................................................................................................... 42
  3.4 Selection of Input value ................................................................................................. 47
  3.5 Oracle Development Strategy ....................................................................................... 48
  3.6 Results Analysis Techniques ......................................................................................... 49
Chapter 4 Results and Analysis ............................................................................................ 52
  4.1 Full and partial round trip path coverage ..................................................................... 53
  4.2 Mutation score analysis for test suites ......................................................................... 57
    4.2.1 Studying the effect of the number of paths on the mutation score ....................... 64
4.2.2 Studying the effect of the number of transitions on the mutation score ... 65
4.2.3 Studying the effect of different algorithms on the mutation score ... 66
4.2.4 Studying the effect of round trip coverage percentage on mutation score ... 68
4.3 Mutation score analysis for test cases (path level) ... 69
   4.3.1 Impact of complete path length (Size) on the mutation score ... 69
   4.3.2 Impact of path type on the mutation score ... 74
   4.3.3 Impact of self-transitions on the mutation score ... 76
   4.3.4 Impact of prefix on the mutation score of the path covering RTPs ... 78
   4.3.5 Impact of events on the mutation score ... 81
   4.3.6 Impact of the repeated use of events on mutation score ... 84
4.4 Fault detection across different fault types ... 86
4.5 Analysis of Code Coverage ... 89
4.6 Summary ... 91

Chapter 5 New Algorithm ... 93
   5.1 The new algorithm ... 93
   5.2 Evaluating the new algorithm ... 94

Chapter 6 Threats to Validity ... 102

Chapter 7 Conclusions ... 103

References ... 105

Appendix A Depth-First and Breadth-First Algorithms ... 108
Appendix B CruiseControl Class Diagram ... 113
Appendix C Transition trees for OrderedSet ... 114
Appendix D Transition Trees for CruiseControl ... 119
Appendix E Transition Trees Covering Round Trip Paths Fully ... 122
   E.1 Sequence of all round trip paths for CruiseControl ... 122
   E.2 Sequence of all round trip paths for OrderedSet ... 125
   E.3 Tree Structure of all round trip paths for OrderedSet ... 126
   E.4 Tree Structure of all round trip paths for CruiseControl ... 127
Appendix F Random Test Suites ... 128
   F.1 Sequence of random test suite for CruiseControl (6 L1, 6 L2, 4 L3, 13 L4) ... 128
   F.2 Sequence of random test suite for OrderedSet (11 L2 path, 10 L3 path) ... 130
Appendix G Transition trees of the New Algorithm ... 132
Appendix H Additional Hypotheses Analysis ................................................................. 134
   H.1 Mutation score analysis for test suites for each case study .............................. 134
Appendix I Mutants Code Sample .............................................................................. 135
List of Tables

Table 1 Comparison between case studies ................................................................. 32
Table 2 Characteristics of paths in case studies trees .................................................. 41
Table 3 Classification of Mutation Operators ............................................................ 43
Table 4 Number of Timeout Mutants ........................................................................... 45
Table 5 Summary of mutant creation .......................................................................... 45
Table 6 Characteristics of the generated transition trees .............................................. 54
Table 7 Mutation Scores for transition trees for CruiseControl and OrderedSet ........ 57
Table 8 Mutation scores descriptive statistics of test suites ........................................ 60
Table 9 Cost descriptive statistics of test suites .......................................................... 62
Table 10 Descriptive statistics for two sample t-test for test suites mutation scores ...... 63
Table 11 RTP coverage in transition trees (percentages) .............................................. 68
Table 12 Mutation Scores for paths according to their length (CruiseControl) .......... 70
Table 13 Average mutation scores for paths according to their length (OrderedSet) .... 71
Table 14 Random test score compared to all round trip score ...................................... 73
Table 15 Mutation Scores for paths according to path type with respect to length (CruiseControl) ........................................................................................................ 74
Table 16 Impact of path type on the mutation score - Unpaired t-tests results .......... 75
Table 17 Mutation scores for paths according to path type with respect to length (OrderedSet) ........................................................................................................ 76
Table 18 Impact of self-transition on mutation score (CruiseControl)........77
Table 19 Impact of self-transition on mutation score (OrderedSet) ..........77
Table 20 Prefix and RTP scores according to paths’ length (CruiseControl)........79
Table 21 Prefix and RTP scores according to their length (OrderedSet) .......80
Table 22 score of same length paths with different events (CruiseControl) ..........83
Table 23 event code coverage (OrderedSet)....................................................83
Table 24 Classification of transitions according to their event........85
Table 25 Effect of event frequency on path score (CruiseControl) ...............86
Table 26 Percentage of detected and live mutants per mutation operator ..........87
Table 27 Code coverage percentages of test drivers (OrderedSet)...............90
Table 28 Code coverage percentages of test drivers (CruiseControl) ..............90
Table 29 Comparison of Characterizes and mutation scores between DFS/BFS trees and the new algorithm trees .................................................................95
Table 30 mutation score of long paths for the new algorithm trees ................97
Table 31 shared percentage of covered items between new algorithm trees, DFS/BFS trees and full coverage .................................................................98
Table 32 Percentage of detected mutants for the new algorithm trees ..............99
Table 33 Legend for transition trees ...............................................................114
Table 34 Descriptive statistics of t-test for mutation scores for each case study ....134
Table 35 Sample Mutants Code .................................................................135
List of Figures

Figure 1 (a) Finite state design (b) Testing tree (adapted from [5]) ........................................... 20
Figure 3 (a) UML state machine (b) DF transition tree1 (c) DF transition tree2 (d) BF transition tree1 ........................................................................................................................................................................................................................................ 23
Figure 4 CruiseControl State Diagram ............................................................................................... 33
Figure 6 OrderedSet State Diagram ..................................................................................................... 35
Figure 8 Finite state machine for OrderedSet given the selected configuration ..................................... 37
Figure 10 Mutants distribution across mutation operators (CruiseControl) ........................................... 46
Figure 12 Mutants distribution across mutation operators (OrderedSet) ............................................. 46
Figure 14 Mutants distribution across classes (CruiseControl) ............................................................. 46
Figure 16 Mutants distribution across methods (OrderedSet) ............................................................... 47
Figure 18 one tailed t-test ..................................................................................................................... 63
Figure 19 Correlation between number of paths and mutation scores ................................................. 65
Figure 21 Correlation between number of transitions and mutation scores ......................................... 66
Figure 23 mean difference plot for scores of trees generated by different algorithms ......................... 67
Figure 25 Scattered diagram showing positive correlation between RT coverage and mutation score (OrderedSet) ........................................................................................................................................................................ 69
Figure 27 Histogram of the path Length Variations ............................................................................. 72
Figure 29 Scattered diagram showing positive correlation between path length and path mutation score ........................................................................................................................................................................ 72
Figure 31 Scattered diagrams showing the effect of unit increase of the prefix and RTP portion on the mutation score of the path ................................................................. 80
Figure 33 Correlation between code coverage percentages and mutation score .......... 91
Figure 35 New Algorithm for generating transition trees ......................................... 94
Figure 37 Unified scale distribution histograms of mutation scores of all trees .......... 101
Figure 39 Distribution histograms of number of paths generated from all trees ......... 101
Figure 41 Algorithm for generating transition trees using depth first .................... 110
Figure 43 Algorithm for generating transition trees using breadth first ................. 111
Figure 45 Class Diagram (UML state machine traversal) ..................................... 112
Figure 47 Class Diagram (CruiseControl) ................................................................. 113
Figure 48 Transition tree 1 (OrderedSet) ................................................................. 115
Figure 49 Transition tree 2 (OrderedSet) ................................................................. 115
Figure 50 Transition tree 3 (OrderedSet) ................................................................. 116
Figure 51 Transition tree 4 (OrderedSet) ................................................................. 116
Figure 52 Transition tree 5 (OrderedSet) ................................................................. 117
Figure 53 Transition tree 6 (OrderedSet) ................................................................. 117
Figure 54 Transition tree 7 (OrderedSet) ................................................................. 118
Figure 55 Transition tree 1 (CruiseControl) .............................................................. 119
Figure 56 Transition tree 2 (CruiseControl) .............................................................. 120
Figure 57 Transition tree 3 (CruiseControl) .............................................................. 121
Figure 59 Tree Structure of all round trip paths (CruiseControl) ............................. 127
Figure 60 one of the possible generated transition trees using the new algorithm (OrderedSet) ................................................................. 132
Figure 61 one of the possible generated transition trees using the new algorithm (CruiseControl).
Chapter 1: Introduction

1.1 Background

Model-based software testing (MBST) has been evolving in the literature since late 1970s. MBST is a software testing approach in which test cases can be derived systematically from software models. Since a large amount of today software development makes some use of the Unified Modeling Language (UML), the UML is an obvious choice for MBST of the system under test (SUT).

MBST has many advantages over other software testing approaches. One of the main advantages of MBST is that it supports the reuse of software specification and design artefacts. These design artefacts, including the UML models, play different vital roles in the software testing process. For example the UML models can be used to systematically generate test data that can be used to automate many of the testing process steps. They can also be used to identify values to be expected when test cases execute.

Another advantage of MBST is that it helps test designers follow well defined systematic testing strategies that can increase fault detection chances and hence increase the effectiveness of the test. In addition, the use of MBST approach to drive test suites from models is not limited to system testing; On the contrary, MBST is commonly applied to integration or unit testing. Moreover numerous experiments with various forms
of software models, both structural and behavioural were used for testing and have reported positive results [1], [2], [6].

Because of the above mentioned advantages the interest in the MBST approach is ever growing and improvements to the existing proposed techniques for MBST are still required.

The UML state machine is one of the available UML diagrams. It is one of the UML diagrams that can provide a clear description of the behaviour of the state-dependant classes of the object-oriented systems. Because of this efficiency the UML state machine diagram has been extensively used in the industry especially for reactive systems (e.g. telecommunication and automotive systems) [1]. This wide use of UML state machine nominates the state machine as the best candidate to be used as a base for defining different MBST techniques.

A variety of testing techniques were proposed based on UML state machine diagrams. They lead to test cases that are sequences of states and transitions extracted from the UML state machine.

It was shown by previous studies [1] that the fault revealing capability of the techniques based on UML state machine is the best for unit testing among other techniques that are based on other models, including sequence diagrams, regardless of the used technique. The UML state machine based techniques have a high capability particularly at revealing unit level faults for embedded systems [1]. Because of this superior fault revealing capability of UML state machine based techniques, these techniques remain one of the most commonly used techniques for testing in spite of the
high cost associated with it. The high cost is due to the large number of generated test cases depending on the complexity of the SUT.

One of the several techniques that were proposed based on UML state machine diagrams is using a transition tree [3], [4], which is an adaptation of an earlier technique defined for finite state machines [5]. A transition tree can be generated by traversing the UML state machine (graph) either depth first or breadth first. By using any of these algorithms, different possible alternative transition trees can be generated from the same UML state machine. The generated tree is composed of nodes that represent the SUT states and edges that represent the transitions. Each path of the generated transition tree, from the root of the tree to a leaf node, represents a test case that consists of a legal sequence of events that transfer the system form one state to another.

Another technique for UML state machine is using round trip paths [3], [4]. The round trip path represents sequences of transitions that starts and ends at the same state without other repetitions of states. Binder [3] suggests that deriving, and testing from, a transition tree is equivalent to testing round trip paths.

Past research proved that transition trees cover some round trip paths entirely while other round trip paths are only covered in a piecewise manner [2]. Therefore Binder makes the hypothesis that testing round trip paths in a piecewise manner is equivalent to testing them fully. Another hypothesis is that the algorithm being used to traverse the state machine graph and create the transition tree has no impact on the testing technique. These hypotheses lead us to identify five different but related research questions concerning trees and round trip paths that have not been studied yet:
• Research Question 1: Does the full coverage of all round trip paths leading to better fault revealing capability than the partial coverage presented in the form of transition trees?

• Research Question 2: Can the full coverage of all round trip paths detect certain types of faults that cannot be detected by partial coverage presented in the form of transition trees?

• Research Question 3: Can the use of different algorithms used to generate the tree affect the fault revealing capability of the tree, knowing that the trees generated either by the breadth or depth-first algorithms are equivalent in the sense that they partially cover the round trip paths?

• Research Question 4: Can we identify paths characteristics (for instance the presence of loops) that can negatively affect the path's effectiveness?

• Research Question 5: Is it possible to reduce the cost, measured for instance in terms of paths number and their size, without reducing the efficiency of the test suite?

The increased demands for new techniques that utilize different UML models resulted in increased efforts to evaluate and modify the current techniques and propose new ones. These demands justify our plan to study and understand some of the available UML model based testing techniques. In addition, designing new traversal algorithm that eliminates the shortcomings of previous techniques will be an important contribution to improve testing results.
1.2 Main contributions

In this research we discuss a model based testing approach for testing state-based software systems described by UML state machine. Through multiple empirical experiments, we attempted to evaluate the cost and effectiveness of different flavours of one testing technique based on UML state machines. The cost was measured in terms of number and length of the generated test cases (paths), and the effectiveness was measured in terms of number and type of detected faults in the SUT.

In order to verify the hypothesis stated in the Introduction and study the research questions specified in Section 1.1, we had to generate multiple test suites using different test techniques. To derive these test suites from the UML state machine we had to design and implement different algorithms that automatically generate the test suites, and then execute these test suites against the SUT. Finally we had to evaluate and compare the results gained from each of the executed test suites through quantitative and qualitative analyses, aiming to uncover test effectiveness impediment, and thus improve the state based testing technique. We also proposed a new, more effective algorithm for constructing transition trees that requires minimum cost.

1.3 Thesis organization

Chapter 2 provides a review of the literature. Chapter 3 provides a detailed description of the experimental design. Chapter 4 analyses the results gained from the experiments. Based on this analysis a new algorithm is proposed in Chapter 5. Chapter 6 discusses threats to validity. Chapter 7 provides our conclusions and recommendations for future research.
Chapter 2: Related Work

The proposed techniques for UML state machine testing are usually based on one or more coverage criteria that allow the test engineer to design a controlled process for software testing. These coverage criteria measure the degree of coverage achieved by a set of test cases. For example a coverage criterion for state based testing may require covering all transitions, or all states of the UML state machine and other criteria can be found in the literature (e.g., [3], [7]). Round tripping is one of the valid and most reliable coverage criterion that helps the test engineer define sequences of state-transitions that form a complete practical use of the system, as it covers all events – responses paths in a state machine diagram. A round trip path is a path that starts and ends at the same state without other inner loops: the only state allowed to repeat is the start (and end) state as they only appear at the beginning and end of the round trip path. In this chapter we will first (section 2.1) discuss previous work proposed to cover a state machine other than the ones based on round trip paths. Then (section 2.2) we discuss the techniques that cover the round trip paths (in a piecewise manner) such as the “n-switch coverage” proposed by Chow [5] and the “Transition tree” proposed by Binder [3], and compare them to the complete full coverage of all round trip paths for a UML state machine graph. Note that although it has been discussed that transition trees do not cover fully round trip paths [2], it has not been studied yet to what extent the round trips are covered by transition trees.
In section 2.3 a sample of previous work that compares different UML model based testing techniques is presented. Some experimental studies that compare the cost-effectiveness of strategies previously proposed for UML state machine testing are presented including round-tripping and transition trees.

2.1 State-Based Criteria other than Round trip path coverage

In any non-trivial state machine, the number of possible sequences of events is too large (if not infinite) because of loops. That is why a number of strategies have been proposed for state-based testing such as all states, all events, all transitions, all transition pairs, full predicate and many others [3], [7].

The “all transitions” criterion requires covering every transition in the UML state machine. The “all transition pairs” criterion requires the test suite to contain tests that traverse each pair of adjacent transitions. The full predicate coverage criterion requires that test cases cause each clause in every guard condition to be true.

2.2 Round trip path coverage

We will focus on round trip strategy as it is the pivot subject for this research. We will shed light on previous techniques that cover the round trip paths, and explain how they are partially covering round trip paths. We will discuss how the full coverage of round trip paths can be achieved.

The “n-switch” coverage criterion [5] is one of the leading techniques that include partial coverage for round trips. It requires covering sequences of consecutive transitions in the state machine graph of length “n+1”, i.e., paths of length n+1 (i.e., with n+1 transition). So 1-switch coverage corresponds to transition pairs coverage. The “n-switch
set” for a transition is the set of all n-switches that start with this transition [5]. For example by looking at the state machine in Figure 1 (a), “1-switch” coverage leads to the set of test requirements (to be covered) \{ab, ac, bb, bc, cd, da, de, ed\}, the “1-switch set” for transition a is \{ab, ac\} and the “1-switch set” for all the transitions is \{ \{ab, ac\}, \{bb, bc\}, \{cd\}, \{da, de\}, \{ed\} \}.

![Figure 1](image)

**Figure 1** (a) Finite state design (b) Testing tree (adapted from [5])

The first significant attempt to cover round trip paths, though in a piecewise manner, was presented by Tsun S. Chow in 1978 [5]. The proposed “n switch set cover” testing strategy is a modification to the previously described “n-switch” criterion. The strategy depends on two kinds of primitives: the inputs from the environment outside the software (stimuli) and the events caused by the software system as a response to the stimuli (operations). The strategy assumes that the machine is minimal, completely specified, starts with a fixed initial state, has a finite number of states and that every state is reachable. The “n-switch set cover” can be obtained by finding test sequences in which all transitions sequences in the “n-switch set” are reached via the same shortest path/prefix starting from root state [5]. Again by looking at the state machine in Figure 1 (a), the “1-switch set cover” for the “1-switch set” of transition b is \{abb, abc\}, and the “1-switch set cover” for the whole state machine is \{ \{ab, ac\}, \{abb, abc\}, \{abcd\}, \{eda,}
ede}. Note that since a and e are outgoing transitions of the starting state 1, the shortest prefix is empty. It is often the case that paths in the “n-switch set cover” for some transition are sub-paths of paths in the “n-switch set cover” for other transitions, thereby allowing the test engineer to reduce the number of test cases. In our example, the “1-switch set cover” ab for transition a is a sub-path of the “1-switch set cover” abb of transition b, the “1-switch set cover” abc of transition b is a sub-path of the “1-switch set cover” abed for transition c. The set of paths / test cases can therefore be reduced: { ac, abb, abcd, eda, ede}.

The “n-switch set cover” can be obtained from a testing tree such as the one in Figure 1 (b). The generation of these trees was proposed by Chow in 1978 and named the “W-Method”. The W-method is one of the most general and widely used method for generating tests from a finite state machine.

The generation of the testing tree proceeds as follows. The root node of the tree is the starting state of the state machine. The root node is considered the current node. A branch from the current node to a successor node is drawn for each state reachable form the state represented by the current node, and the branch is labeled after the transition between these two nodes. The successor node is considered a terminal node if its label is the same as a non terminal node somewhere else in the tree (the node was branched before) or it represents a final state in the state machine, otherwise the node is considered the current node and the procedure repeats. The test sequences are then paths in the tree from the root node to a leaf. It is worth noting that different traversal algorithms of the graph representing the state machine can lead to different trees.

In this thesis we can classify these tree paths as follow:
• Prefix followed by a complete round trip path: for instance (Figure 1), path ab is composed of a prefix (a) and a round trip path (b).

• Only a round trip path (with no prefix): for instance (Figure 1), path ed is a round trip path.

• Simple path, i.e., a path that is not one of the two above: for instance (Figure 1), path ac is a simple path.

It is then clear that in general the transition tree does not necessarily cover the round trip paths fully: e.g., path acd is a round trip path that is not covered by the transition tree (Figure 1). In other words, the round trip paths are in general only covered in a piece-wise manner: ac and d are pieces of round trip path acd that are covered in the tree of Figure 1 b.

Chow’s W method was subsequently adapted by Binder and denoted as round-trip path testing. It consists in deriving test sequences from transition trees generated by traversing the UML state machine diagram using a depth first algorithm (DFS) or a breadth first algorithm (BFS) with the termination criteria defined above. Even one algorithm, for instance DFS may result in different trees depending on the order the outgoing transitions of a state are considered in the implementation of the algorithm. Binder’s adaptation also accounts for the hierarchical nature of UML state machines (e.g., composite states), and it accounts for guard conditions that are complex expressions. It also includes the notion of sneak path testing: a sneak path exists when the state machine is not fully specified; for each unspecified behavior (e.g., the state machine in Figure 1 does not specify the behaviour in state 3 when stimulus α is received) a sneak
path is a path from the initial state to the starting state of the unspecified transition, followed by the stimulus of the unspecified transition.

Although testing from a transition tree is one of the most commonly used techniques, the technique has some weaknesses and unclear points that need to be thoroughly investigated to enable test designers to make better decisions since a wrong decision could significantly affect the quality of tree paths and consequently the quality of the generated test suite. These issues are: the algorithm used to traverse the UML state machine, the order in which successor nodes are used to branch, the impact of the prefix to the round trip path, and the extent to which a tree covers round trip paths fully.

![Diagram](image)

**Figure 2** (a) UML state machine (b) DF transition tree1 (c) DF transition tree2 (d) BF transition tree1

To clarify these issues three trees were generated from the UML state machine in Figure 2 (a): the first and second trees Figure 2 (b), (c) were generated using a depth first algorithm but with different successor node selections, the third tree Figure 2 (d) was
generated using a breadth first algorithm. Node 2 is circled twice to illustrate different successor node selections: the difference between Figure (b) and (c) is that, from tree root node 1, we can decide to develop the tree either from node 2 after transition a (Figure (b)) or from node 2 after transition b (Figure (c)); Figure (d) shows that another tree (not shown) is possible if we decide to develop the tree from node 2 after transition b. Each of these trees results in different test suites:

Tree1 (using DFS) paths = \{ad, aeg, aehj, aehi, af, b, c\}

Tree2 (using DFS) paths = \{a, bd, beg, behj, behi, bf, c\}

Tree3 (using BFS) paths = \{ad, aeg, aeh, af, b, cj, ci\}

By looking at the UML state machine, Figure 2 (a), we can directly extract all the 23 round trip paths: ad, bd, ci, afi, bfi, aehi, behi (for state 1), da, db, fia, fib, ehia, ehib (for state 2), g, hiae, hibe, hj (for state 3), ic, jh, iaf, ibf, iaeh, ibeh (for state 4).

By looking at the previous result we can notice that:

1. Different algorithms generate trees that differ regarding the number of generated paths (this is not the case in Figure 2 but we will illustrate this situation with our case studies), the length of the paths (length up to 4 in trees (b) and (c), length up to 3 in tree (d)), and the type of these paths using the classification above (the number of covered round trip paths in tree (b) is 3, in tree (c) is 4, and in tree (d) is 3). However, they are all equivalent in the sense that they produce adequate test suites for the criterion. Neither Chow nor Binder discuss the issue of the selection of a traversing algorithm, though it has been shown in [6] that different trees have different effectivenesses at finding faults. Note that no systematic study of this phenomenon is reported in [6].
2. The choice of the next successor node to be branched also has an impact: Figure 2 (b) and (c) show different tree paths; though, again, the trees are equivalent in the sense that they produce adequate test suites for the criterion. Chow and Binder did not mention this potential issue.

3. The algorithms tend to use the shortest prefix possible to reach a round trip path, and then trigger the full round trip path or part of it. In case there is more than one shortest prefix to reach a given round trip, do they have the same effectiveness (at finding faults)? Are there any special criteria to identify the best prefix? For example the round trip {g} can be reached using prefix ae, be, or ci.

4. The algorithms tend to use the same prefix when this prefix allows us to trigger several round trip paths. However, choosing a different prefix for each round trip path would result in more different sequences and therefore, perhaps effectiveness. For example for the round trip paths that start from node 3 we can use two different prefixes: aehj, or cjg.

5. Many round trip paths are not fully covered by each tree. For example, the tree of Figure (b) only covers fully four of the 23 round trip paths, specifically ad, aehi, g, and hj. The tree of Figure (c) only covers fully four of the 23 round trip paths, specifically bd, behi, g, and hj. The tree of Figure (d) only covers fully three of the 23 round trip paths, specifically ad, ci, g. The trees cover varying number and sets of round trip paths fully, and most of the long paths are never covered fully.

6. How significantly can a change of algorithm termination criterion affect the cost and effectiveness (fault detection) of generated transition trees?

All these are valid issues to be investigated.
2.3 Previous experiments with state based testing

We classified previous experiment studies into three types. The first type is the empirical studies and industrial practices that tried to assess the fault revealing capabilities of the tests generated by the different UML models and associated criteria. The second type is the studies that tried to compare the effectiveness of different coverage criteria for UML state machine based testing. The third type is the studies that attempted to modify and improve existing UML state machine based techniques, and some were able to propose new test generation strategies aiming to enhance their fault revealing capabilities.

In this section a selected work sample from the three types is presented. The sample covers a variety of academic and industrial experiments that tried to evaluate, compare and enhance state base testing criteria. Some experiments used hand-seeded faults to evaluate the fault revealing effectiveness of test suites; others used automatically generated and seeded faults (mutants), while industrial experiments were performed on real faults.

One of the experiments of the first type that aims to compare different UML models capabilities for testing purposes was done on a single project [1]. The faults were manually inserted into the implementation of a typical cell phone software. The application was modeled by five UML state machines with guard conditions and actions\(^1\) on transitions as well as six sequence diagrams with 37 alternatives fragment. The research compares the effectiveness of 81 test cases generated by hand from UML state machine using the full predicate criterion [7] and 43 test cases generated by hand from

\(^1\) Actions represents what happens when the transition is taken.
UML sequence diagrams using Message sequence path coverage [8]. After running each of the two test sets generated from each diagram type on the implementation, it was found that the UML state machine test set was better at revealing unit level faults while the sequence diagram test set was better at revealing integration level faults.

Another experiment of the first type presents an approach to generate test cases automatically from UML sequence diagrams where Object Constraint Language (OCL) is used in message guard conditions. The approach contracts a scenario tree from the sequence diagram and acquires the message path from the tree for a certain scenario. Then OCL is used to describe the pre- and post conditions for each attribute of the objects related to the scenario. The test cases generated using this approach achieve message paths coverage and constraint attribute coverage. The results of experiment show high performance [9].

Another experiment of the first type investigates the fault detection capability of UML state machine based testing of class clusters with state-dependent behaviour [10]. The study compares UML state machine based testing with the common practice of using code coverage analysis to derive test suites. UML state machine based testing does not appear more effective at detecting faults than simple structural testing, and that structural testing is not more effective than state based testing. However, the results show that UML state machine and structural testing techniques are complementary in terms of the faults they detect, which suggests that the two strategies should be combined in practice. As it is common, the authors recommends using UML state machine testing first, since it can be planned and prepared early during the development process (even before the code is
ready), and then complement it by control flow analysis, to reach the appropriate code coverage.

Another empirical study, that belongs to the second type of studies, tried to assess the cost-effectiveness of four of the most referenced adequacy criteria for state based testing [11], namely “all transitions”, “all paths in transition tree”, ”all transition pairs”, and “full predicate”. Results show that the transition tree is more effective than all transitions and about as expensive, but less effective than all transition pairs or full predicates (which are much more expensive). Transition tree therefore seems to be a good alternative, in terms of cost and effectiveness, between all transitions and the other two, if no very high fault detection rates are needed. The authors observe that different trees can have different effectivenesses, though this is not systematically studied, and they advice that it is better to select the tree that exercises the code in the most realistic and complete manner, which could be achieved by applying a procedure similar to the one described above [11] or by selecting a tree that exercises the system in a as realistic as possible way.

Another experiment attempts to compare UML state machine testing to random testing [12]. The study shows that among the different well established criteria proposed for UML state machine, round trip path testing still remains a good compromise [12].

Many studies of the third type attempt to refine the round-trip path criteria as this is one of the most referenced criteria. One study attempts to combine the round trip path criteria with another black box testing techniques, specifically Category-Partition [13] for the selection of input data when triggering transitions [12]. The authors successfully combine the two techniques to obtain a better effectiveness at finding faults, at the expense of a reasonable additional cost. Another study addresses the problem of selecting
a transition tree among a number of possible trees derived from a state machine [14]. Out of many possible generated trees the study was able to identify and select the greatest fault detection tree by extracting, and using, the data flow information contained in the trees. The experiment concludes that both “definition-use pairs” and “definitions”, two well-known data flow criteria (which were adapted to the context of transition trees), are good indicators of fault detection effectiveness of a transition tree: The larger the number of “definition-use pairs” or “definitions” the tree covers the higher its fault detection effectiveness. Since identifying “definitions” is easier and less costly than identifying definition-use pairs, the authors recommend to use the former in practice. The authors also observe that different trees cover different definitions. To improve the performance of a transition tree, they therefore propose an improved tree construction whereby the best transition tree (in terms of definition coverage) is complemented with path from other trees that cover the missing definitions.
Chapter 3 : Experimental Design

The hypotheses we mentioned at the beginning of this thesis (Chapter 1) led us to formulate five questions that we believe are worth investigating and that we recall here:

- To what extent does a transition tree covers round trip paths, i.e., how many round trip paths does a transition tree covers? And does this depend on the general strategy used to build the transition tree, i.e., depth-first traversal or breadth-first traversal?

- How does covering round trip paths fully or through a transition tree differ in terms of effectiveness at finding faults? And does the difference, if any, depend on the general strategy used to build the transition tree, i.e., depth-first traversal or breadth-first traversal?

- How does covering round trip paths fully or through a transition tree differ in terms of the types of faults they detect? And does the difference, if any, depend on the general strategy used to build the transition tree, i.e., depth-first traversal or breadth-first traversal?

- How does covering round trip paths fully or through a transition tree differ in terms of cost? And does the difference, if any, depend on the general strategy used to build the transition tree, i.e., depth-first traversal or breadth-first traversal?
- Do the results depend on factors such as the size of test suites, the types of paths the test cases contain (simple path, complete round trip path ...)?

In order to answer these questions, two case study systems were chosen from two different domains of application where state-based behaviour can be encountered: one system is a reactive system, specifically a CruiseControl, whereas the other is a data structure, specifically an ordered set of integers. For each case study system we built several test suites: one test suite, manually derived, to fully cover all the round trip paths in the state machine, several test suites, automatically derived, corresponding to different traversals of the state machine graph (e.g., depth vs. breadth first) to obtain transition trees.

To study the effectiveness at finding faults of those test suites, we seeded mutants, using standard mutation operators. This is a standard procedure in the academic testing community when one does not have access to real faults [15], [16]. The mutation score, that is the percentage of detected (or killed) mutants to the total number of seeded mutants, is the measure commonly used to compare effectiveness.

To study the cost of those test suites, we used surrogate measures of cost, which, again, have been used often in the testing community. Specifically, we considered the number of test cases in test suites, the cumulative length of test cases in test suites (the length of a test case being the number of transitions it triggers).

For each test case of each test suite we created test drivers. This allowed us to automatically perform all our experiments. We also used a tool to automate the creation (and compilation) of mutants.
The rest of this Chapter discusses those aspects in a number of sections. Section 3.1 presents the two case study systems. Section 3.2 provides details on the generation of test suites: to fully cover round trip paths; to build every possible depth-first and breadth-first traversal transition tree. Section 3.3 discusses the generation of mutants. Section 3.4 discusses the selection of input values for the OrderedSet case study. Section 3.5 discusses our strategy for building test oracles. Finally, section 3.6 presents the analysis techniques we used to answer the abovementioned questions.

### 3.1 Case Study Systems

Two systems were selected from different areas: the CruiseControl system represents reactive systems and the Ordered set system represents data structures. These systems were also selected as their state machine diagrams lead to multiple transition trees when using depth-first and breadth-first traversal algorithms (Section 3.2). We believe that the high connectedness and cyclic nature of the statechart may lead to obtaining the same trees regardless of the (two) traversal algorithms (this comes from our observation of results with CruiseControl).

Table 1 details some characteristics of the two case study systems, in terms of number of states, events, transitions, and the number of code lines in all their Java classes. The table shows that guard conditions in OrderedSet were omitted, which is further discussed in Section 3.1.2.

<table>
<thead>
<tr>
<th></th>
<th>CruiseControl</th>
<th>Ordered Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of states</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Transitions</td>
<td>29</td>
<td>33</td>
</tr>
<tr>
<td>Events</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Conditions</td>
<td>Not considered</td>
<td>Omitted</td>
</tr>
<tr>
<td>Number of classes</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Number of lines of codes (Without comments)</td>
<td>280 LOC</td>
<td>250 LOC</td>
</tr>
</tbody>
</table>

*Table 1 Comparison between case studies*
3.1.1 CruiseControl Case Study

The CruiseControl system simulates a car engine and its cruising controller. It is implemented in Java and it contains six classes; CruiseControl, Controller, CarSimulator, SpeedControl, CruiseDisplay and CarSimulator—the class diagram is in Appendix B. The first four classes form a class cluster whose behaviour is described by a state machine: Figure 3. Notice from the diagram that events have no parameters and no guard conditions: the state machine is a finite state machine.

![CruiseControl State Diagram](image)

**Figure 3 CruiseControl State Diagram**

3.1.2 OrderedSet Case Study System

The ordered set case study represents a bounded, ordered set of integers (Figure 4). It is implemented in java in a single class named OrdSet. When an OrderedSet is first created, its size gets initialized. The size of an OrderedSet represents slots that can be
used to add integers to the set. The size should be at least equal to the minimum set size and it should not exceed the maximum set size. The size of an OrderedSet is always a multiple of min_set_size. A single element can be added or removed from the set. If a new element is added to a full OrderedSet, the OrderedSet can resize itself, thereby allowing further additions. The number of resizes allowed is a constant number.

Attempting to resize the set more than max_accepted_resizes times or for a size that exceeds the maximum set size would cause an overflow (overflow exception). An attempt to add or remove an element from the set after an overflow raises an overflow exception.
As can be noticed from Figure 4, the events have parameters (the integers to add and remove) and have guard conditions. This is a problem for transition tree generation since the depth-first and breadth-first traversal algorithms do not account for guards and input parameters—instead they only rely on the structure of the diagram/graph—and therefore can generate unfeasible paths, i.e., unfeasible test cases. Binder adapted the construction of transition trees to account for guard conditions. However, his solution is incomplete,
manual, and makes additional hypothesis. We therefore do not use his adaptation. To prevent the problem of generating unfeasible paths, we need to transform the UML state machine of Figure 4, which is in fact an extended finite state machine, into a finite state machine, i.e., we need to transform the abstract states of Figure 4 into concrete states: e.g., we need to transform the Partially Filled state into as many concrete states as necessary representing a set with 1, 2, 3, ... element(s). This is impossible in general since the maximum size of the OrderedSet is an initialization parameter. However, if we fix that maximum size as well as other initialization parameters to reasonable values, we can do the transformation.

We believe that such a transformation, if performed with care (e.g., avoiding obvious simplistic initialization parameter values such as a maximum set size of 1), will lead to a finite state machine with a reasonable number of states, while not limiting the conclusions we may draw from our case study. This is also a transformation that has already been performed in the past [11].

In our study we therefore set the min_set_size to 2 (the minimum size is 2 and this is also the number of new slots that are added when resizing), the max_set_size to 6, and the max_accepted_resizes to 2. Figure 5 is the finite state machine, resulting from transforming the abstract states of Figure 4 into concrete states given this configuration. It now shows eight concrete states, including three filled states (in our configuration, the OrderedSet can resize only twice).
Figure 5 Finite state machine for OrderedSet given the selected configuration

In Figure 5, there are three input values that distinguish add and remove events for the OrderedSet: x, n, and val; where x is an integer, n is a new integer that is not included in the set, and val is an integer that is already included in the set.

3.2 Test Suite Generation

In this section we first discuss the generation of depth-first and breadth-first transition trees (Section 3.2.1), and then the generation of test cases covering round trip paths fully
(Section 3.2.2). We then identify and discuss different kinds of test cases that we obtain in our test suites, a classification that will be useful in our case studies (Section 3.2.3).

3.2.1 Algorithm for generating transition trees

Chow's transition tree algorithm [5], adapted by Binder [3] is to traverse the state model and construct a transition tree that includes all transitions in the state machine in such a way that the traversing along a path stops whenever the state encountered is already present in the tree. More precisely, Binder's description of the algorithm is the following: the initial state is the root of the tree; an edge is drawn for each transition out of the initial node, and the resultant state of this transition is represented by a node in the tree; if the state this node represents is already drawn somewhere in the tree, or is a final state of the state machine, this node is considered terminal and no more transitions are drawn from it. The drawing of edges out of non-terminal nodes then proceeds similarly to the root node, until all the leaf nodes of the tree are terminal.

In essence, this algorithm is a breadth first traversal of the graph representing the state machine. There are therefore at least two families of transition tree generation algorithms since, alternatively, one can use a depth first traversal: the only step to change is the drawing of edges out of non-terminal nodes. Other types of algorithms can be considered but depth and breadth-first traversal algorithms are the most well known and used ones. Studying other types of algorithms is out of the scope of this thesis.

Either with a depth-first or breath-first algorithm, one can potentially obtain different transition trees as there may be several ways to work on the non-terminal nodes. Indeed, in the algorithm described above, the non-terminal nodes can be considered in different
sequences, thereby possibly leading to different transition trees. This depends on how the traversal algorithm is implemented.

These depth-first and breadth-first algorithms are all equivalent, that is the transition trees they generate are equivalent, in the sense that they cover the round trip path (though in a piece-wise manner). Differences between generated transitions trees, in terms of effectiveness at detecting faults have already been mentioned in the literature [11] study the impact of the transition tree generation algorithm on effectiveness.

We therefore implemented two algorithms, one depth-first and one breadth-first, that generate all possible depth-first or breadth-first traversals of the state machine graph, thereby generating all possible transition trees using a depth-first or breadth-first traversal algorithm. The depth-first and breadth-first algorithms, as well as a design (class diagram) for their implementation are discussed in Appendix A.

Using depth first traversal, the CruiseControl state machine led to three transition trees, while the OrderedSet state machine led to seven trees. Using the breadth first traversal, the CruiseControl led to the same three trees: this is due to the fact that the graph representing the state machine is cyclic; and the OrderedSet state machine led two trees. The trees for OrderedSet and CruiseControl can be found in Appendix C and Appendix D, respectively.

3.2.2 Test Cases Covering Round Trip Paths Fully

We manually created, for each case study system (i.e., each finite state machine), one test suite that covers round trip paths entirely. To do so we first identified all the round trip paths: we started by identifying all the states from which a round trip path can start; then we identified, for each such state, all the round trip paths that it can start.
To obtain a test case for each identified round trip path, we need a complete, feasible path made of a prefix (i.e., a path from the starting state of the state machine to the state starting the round trip path) followed by the round trip path. To select a prefix for each round trip path we manually identified the shortest path that begins with the state machine start state and ends at the beginning of the round trip path. In case we identified several paths with the smallest length, we choose the path that has the highest variety in terms of the events it involves. If this heuristic is not sufficient to distinguish paths, i.e., some paths have the same variety of events, we choose the path that involves the largest number of events that are least used in the rest of the tree (i.e., it involves more events that are not used often in the rest of the tree). Since we deal with finite state machines, these complete paths are feasible. Note that a test suite satisfying the all round trips coverage criterion does not necessary cover all transitions.

Admittedly, different prefixes could lead to different effectivenesses at finding faults. Trying different techniques to build prefixes is not part of our study. The reason is that we are mostly interested in the effectiveness of round trip paths, which we will analyze separately from their prefixes. Additionally, the round trip path test cases do include some round trip path with prefixes that are not necessarily the shortest paths as this depends on the traversal algorithm.

Last, since all the prefixes start from the initial state of the state machine, they can all be displayed in a transition tree rooted at this initial state. Therefore, we consider that our manual generation of test cases that cover round trip paths fully is a transition tree generation algorithm (though it is not automated). In the rest of this thesis, we will
therefore only talk about transition trees and we will make it clear whether they cover round trip path fully or are derived from a depth-/breadth-first traversal.

3.2.3 Classification of Test Cases

The transition trees, either automatically (i.e., depth/breadth first traversal) or manually (i.e., covering round trip paths fully) generated, are characterized by having two types of paths (test cases):

1. Paths that contain a round trip path and which need a prefix; the prefix may only contains one transition, i.e. a call to the constructor, which we refer to as pure round trips; or can be of length at least two before including a round trip path;

2. Simple paths that do not include any round trip.

Recall the three types of paths we have discussed in section 2.2. In our case studies, one type of paths does not exist, specifically, paths that are “only round trip”. This type is not applicable because none of the initial nodes of the state charts of the two case studies starts a round trip.

Table 2 shows the number of each type of paths of each tree for both case studies.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>OrderedSet</th>
<th>BFS</th>
<th>Full coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trees</td>
<td>T1  T2  T3</td>
<td>T4  T5  T6</td>
<td>T7  T1  T2</td>
</tr>
<tr>
<td>Prefix+RTP</td>
<td>19  20</td>
<td>19  18  20</td>
<td>20  19  18  17 21</td>
</tr>
<tr>
<td>Simple paths</td>
<td>6   5</td>
<td>6   6</td>
<td>5   5    6   7   8   0</td>
</tr>
<tr>
<td>CruiseControl</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Algorithm</td>
<td>BFS/DFS</td>
<td>Full Coverage</td>
<td></td>
</tr>
<tr>
<td>Trees</td>
<td>T1  T2  T3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prefix+rtp</td>
<td>23  23</td>
<td>23   29</td>
<td></td>
</tr>
<tr>
<td>Simple paths</td>
<td>2   2</td>
<td>2    0</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>25  25</td>
<td>25   29</td>
<td></td>
</tr>
</tbody>
</table>

Table 2 Characteristics of paths in case studies trees
3.3 Mutant Generation

Mutants are simple changes made to the program (source) code, and are created from a set of mutation operators which describe (at a high level) what those changes involve [17],[18],[19],[20]. Mutants are assumed to emulate typical programmer mistakes/faults, mutation operators are assumed to be representative of the family of faults programmers make. There is empirical evidence that these assumptions hold [19]. Mutants have therefore been used for a long time in experiments similar to the one we perform in this thesis, where different testing techniques are compared.

Despite the fact that generating a good set of mutants is crucial for the success of the experiment and for reaching trustable rich analysis results, the process of seeding faults to create mutants and testing them remains impractical without automated tools because the larger the number of mutants generated and the more variation of mutants types, the more accurate the results [21]. Therefore most of the mutants in this research were generated using the mutation system for Java “MuJava” (408 mutants created by MuJava for CruiseControl and 605 for OrderedSet) and only few were manually seeded with faults (eight mutants manually created for CruiseControl and 4 for OrderedSet). We manually seeded mutants for mutation operator not supported by MuJava: specifically IOD, IOR, and OLR [20], [22]. We used both traditional and class-level mutation operators. They are intended to cover three main areas: the first is the object-oriented features which involve encapsulation, inheritance, polymorphism, and type conversion; the second is syntactical features specific to Java, including insertion, deletion and alteration of keywords, modifiers, variables, constructors and methods; the third is general traditional mutants including different types of operators (arithmetic, relational, conditional, logical, and
assignment operators). Table 3 lists the mutation operators we used: the * indicate manually seeded mutants.

<table>
<thead>
<tr>
<th>Area</th>
<th>Category</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Object-oriented features</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Encapsulation: AMC Access modifier change</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SMC Static modifier change</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inheritance: IOD* Overriding method deletion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IOR* Overriding method rename</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Polymorphism: OLR* Over loading method rename</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Java syntactical features</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Class Level Keywords: JTD This keyword deletion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Modifiers: JSI Static modifier insertion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>JSD Static modifier deletion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Variables: JID Member variable initialization deletion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Constructors: JDC Java-supported default constructor deletion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Methods: EAM Accessor Modifier Change</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Statement Level Arithmetic operators</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AORb Arithmetic operator replacement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Replace basic binary operator with other binary operator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AORs Arithmetic operator replacement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Replace short-cut operator with other unary operator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AOIs Arithmetic operator insertion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Insert short-cut arithmetic operator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AOIu Arithmetic operator insertion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Insert basic binary arithmetic operator</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Relational operators: ROR Relational operator replacement</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Conditional: COI Conditional operator insertion</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Logical operators: LOI Logical operator insertion</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Assignment operator: ASRs Assignment operator replacement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Replace short-cut arithmetic operator with other unary</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AODu Arithmetic operation deletion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Delete unary arithmetic operator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AODs Arithmetic operation deletion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Delete short-cut arithmetic operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COR Conditional Operator replacement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COD Conditional Operator deletion</td>
</tr>
</tbody>
</table>

**Table 3 Classification of Mutation Operators**

The total number of mutants created was 416 for CruiseControl and 609 for OrderedSet.

We observed that some mutants did not compile: eight (resp. 12) mutants did not compile for CruiseControl (resp. OrderedSet). For example JSD mutant where keyword "static" is removed lead to a compilation error because no static reference can be made to

43
a non-static field; another example is an AMC mutant where the access modifier is changed and results in a compilation error because the field is not visible.

We also observed that some mutants crashed during execution; 14 (resp. four) mutants crashed for CruiseControl (resp. OrderedSet).

These mutants, that were either caught by the compiler or crashed (i.e., were too easy to kill) were not considered. Most of these mutants are from class level mutation operators such as JSD, AMC, JDC and IOD, and some from traditional mutation operators such as AORb, AOIs, and LOI.

Equivalent mutants are mutants whose behavior cannot be distinguished from the original program, and are known to exist and be difficult to identify [23], [24], [25], [26]. We considered mutants that could not be killed by any test case in our experiments (not killed by 100% of the test cases in the different trees) as “Very hard” to kill since they are equivalent (given the set of test cases we have) to the original program. This set of mutants contains mutants that are indeed equivalent ones (cannot be killed by any test case one can think of). It can also contain mutants that could be killed, though test cases that would kill them are not in our pool of test cases (from our different trees). Since these mutants would not allow us to see differences between the transition trees we experiment with, they were discarded. In total, 118 mutants for CruiseControl and 98 for ordered set are (considered) equivalent.

We also considered mutants that can be detected by at most 5% of the test cases (i.e., not killed by 99% to 95% of the test cases) as “Hard” to kill. These hard to kill mutants were excluded to avoid having extreme outliers in our analyses (69 mutants for CruiseControl and 78 for ordered set).
These heuristics conforms to what has been done in the past by others [2].

Using these heuristics, we discarded 45% (resp. 29%) of the mutants for CruiseControl (resp. OrderedSet). (These percentages correspond to 187 mutants for CruiseControl and 176 mutants for OrderedSet.)

During execution, mutants that timed out because of infinite loops were considered to be killed mutants. The number of time out mutants for OrderedSet and CruiseControl is shown in Table 4.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>OrderedSet</th>
<th>BFS</th>
<th>Full coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trees</td>
<td>DFS</td>
<td>T1</td>
<td>T2</td>
</tr>
<tr>
<td>Timeout mutants</td>
<td>25</td>
<td>19</td>
<td>24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>CruiseControl</th>
<th>BFS</th>
<th>Full Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trees</td>
<td>DFS</td>
<td>T1</td>
<td>T2</td>
</tr>
<tr>
<td>Timeout mutants</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 4 Number of Timeout Mutants

To summarize, Error! Reference source not found. shows the number of created, discarded and equivalent mutants for the CruiseControl and OrderedSet case study systems.

<table>
<thead>
<tr>
<th>Number of Mutants</th>
<th>CruiseControl</th>
<th>OrderedSet</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of mutants generated by MuJava</td>
<td>404</td>
<td>605</td>
</tr>
<tr>
<td>number of mutants generated manually</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>Total number of generated mutants</td>
<td>416</td>
<td>609</td>
</tr>
<tr>
<td>number of mutants that do not compile</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>number of mutants that crash</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>number of mutants that are considered equivalent</td>
<td>187</td>
<td>176</td>
</tr>
<tr>
<td>- Not killed by any test case (100%)</td>
<td>118 (63%)</td>
<td>98 (56%)</td>
</tr>
<tr>
<td>- Not killed by (99-95%) of the test cases</td>
<td>69 (37%)</td>
<td>78 (44%)</td>
</tr>
<tr>
<td>Total number of discarded mutants</td>
<td>209</td>
<td>192</td>
</tr>
<tr>
<td>Remaining mutants</td>
<td>207</td>
<td>417</td>
</tr>
</tbody>
</table>

Table 5 Summary of mutant creation

Not all of these mutants' operators were applicable to both cases under study. The remaining mutants are distributed across mutation operators for each case study as shown
in Figure 6 and Figure 7. The mutants’ distributions across classes for the CruiseControl case study are shown in Figure 8 and Figure 9 show the mutants’ distributions across methods for the OrderedSet case study because the case study is composed of one class only.

![Graph showing mutants distribution across mutation operators for CruiseControl](image1)

**Figure 6 Mutants distribution across mutation operators (CruiseControl)**

![Graph showing mutants distribution across mutation operators for OrderedSet](image2)

**Figure 7 Mutants distribution across mutation operators (OrderedSet)**
3.4 Selection of Input value

To develop test cases for the OrderedSet we had to use input values. For CruiseControl, events triggering transitions do not have input parameters: the state machine is an FSM. The choice of the input values is important as it can drive the selection of varying control flow paths, and cause a poor code coverage, and thus lower mutation score. For example, suppose the OrderedSet contains the four following integers: [1,2,3,6]. In case integer 7 is added, i.e., add(7), since 7 is greater than 6 (the maximum integer in the set), 7 is simply added to the end. We can force the add function to rearrange the set and add at the beginning of the list, by adding a new value smaller than all the existing values, so that values must be shifted. In case integer 5 is added, i.e., add(5), OrderedSet identifies where to add it (between 3 and 6) and integer 6 has to be pushed once to make room for 5. These two scenarios will trigger very different control flow paths in the code and therefore can potentially reveal different numbers of mutants.
In order to exercise different situations, we choose the input values so that at least one addition of a new value and one deletion of an existing value are made at least once in the beginning of the set, in the middle of the set, and in the end of the set. This setting was possible to achieve since each test driver executing a test suite can perform more than three possible additions of new values and more than three possible removals of existing values.

3.5 Oracle Development Strategy

The oracles of the test cases check the states reached by the SUT after the execution of each transition in the test case. The state reached is compared with the expected behavior/state described by the UML state machine.

Checking the state may mean either checking the abstract state (i.e., the values of the variables that define the abstract state in the state machine) or checking the concrete state (i.e., checking all the variable values) [10]. For instance, in a bounded data structure, the partially filled (abstract) state is defined by the fact that the value of the number of elements variable is between zero (strictly) and the maximum size, where as to define the concrete state one has also to consider the contents of the data structure.

We decided to have an intermediate solution whereby we check the abstract state but also part of the concrete state. The rationale is that checking the abstract state may not be sufficient (e.g., reusing the bounded data structure example above, we would not check that added data are added at the right place) and completely checking the concrete state may be too expensive.

As an example, to check state partially filled (SUT OrderedSet), we check the value of the maximum current set size (should be strictly positive), of the maximum current set
size (should be strictly greater than the actual set size), and of the number of resizes (should be smaller than the maximum number of allowed resizes). For the CruiseControl SUT, checking state Cruising involves the value of the ignition status (should be true), of the controller status (should be cruising), and of the speed control (should be enabled).

Additionally, the oracle compared some variable values with expected ones. For example, in the OrderedSet SUT, if an existing value is removed from the set we check the content of the set to make sure the set does not contain the removed value any more. Another example, for the CruiseControl if a break command was handled we check that the break pedal variable is incremented, or if accelerate was handled we check that the throttle value in incremented.

3.6 Results Analysis Techniques

To help understand, analyze and visualize results we used the statistical tool JMP version 7.0.1. The tool provides a broad range of graphical interface charts and diagrams to display and analyze data. The tool includes a variety of advanced statistical methods for data analysis from which we used paired and unpaired T-test, Wilcoxon rank, and distribution Histograms with multiple plotting options (ex. Plot difference by mean, by row .. etc).

We also used Microsoft Office Excel 2007 to display fit distributions on scattered diagrams with added trend lines.

An online software tool graphpad [27] was used to help us determine the statistical significance of relationship between variables.
To measure code coverage we used CodeCover\(^2\) (Eclipse plug-in). CodeCover supports statement coverage, branch coverage, loop coverage, and condition coverage. It can record the percentage of coverage for individual test cases or complete test suites. The percentage of coverage indicates the portions of the code executed by the test case. The tool also reports on the number of times a code coverage element (e.g., statement, branch) is executed.

To help identify the correlation between data sets and the significance of the relations between variables, we used different statistical analysis techniques [28], [29], [30].

1. **Pearson product-moment correlation coefficient**: is one of the most common correlation coefficient statistical methods, it is used as a measure of the strength of linear dependence/association between two or more variables of the observed data sets. The correlation coefficient of a sample is denoted by \( r \). The sign and the absolute value of a correlation coefficient describe the direction and the magnitude of the relationship between the variables, respectively.

2. **The one sample t-test** is a statistical procedure that is used to know the mean difference between the sample and the hypothesized value of the population mean. If the calculated t-value is smaller than the value of \( t \) retrieved from the distribution table, there is evidence that the mean is not significantly different from the hypothesized value, and we cannot reject the null hypothesis in favor of the alternative.

3. **The dependant t-test for two paired/unpaired different samples** is used to check if the relation between two sets of data—normally distributed—is

\(^2\) [http://www.codecover.org/documentation/tutorials/how_to_complete.html]
statistically significant. The t-test allows us to determine a p-value that indicates the probability of obtaining these results by chance.

4. **Wilcoxon test** (an alternative for the t-test) is used to get the value of t for non-parametric values or small sample data.

5. **Shapiro-Wilk test** is used to check the normal distribution of data sets.
Chapter 4 Results and Analysis

In this chapter we present the transition trees generated using depth first and breadth first algorithms for each of the case studies presented in Chapter 3. The test suites of these trees are compared to the test suites that are manually driven from the UML state machines to cover all possible round trip paths fully. The qualitative analysis performed aim (1) to assess the cost and effectiveness of the test suites, (2) to compare the partial coverage of round trip paths presented in the transition trees to the full coverage of round trip paths, and (3) to identify the factors that affect the mutation score of test cases. Statistical hypothesis testing is applied to help us make decisions to accept or reject hypothesized relations observed from the quantitative sample data, and to determine how likely the assumed relations are prone to reality or due to chance.

We first perform a quantitative comparison of all the trees we generated (section 4.1). The mutation score (effectiveness) of the full and partial round trip coverage is compared in section 4.2, thereby answering Research Question 1 (page 16). The number and types of errors that can be detected by full and partial round trip coverage is investigated in sections 4.4, thereby answering Research Question 2. The algorithm effect on the mutation score, i.e., DFS or BFS, is discussed in section 4.2.3: Research Question 3. The effect of characteristics of tree paths, such as event sequences, input value, state variation, and event types, on the paths score is discussed in section 4.3: Research Question 4. Finally a trade off between cost and size of the round trip path strategy is
presented in section 5.2, where we propose a new tree construction mechanism. Section 4.5 discusses the structural coverage of the tree (black box) test suites.

4.1 Full and partial round trip path coverage

This section presents all possible generated transition trees that cover the round trip paths in a piecewise manner and the trees that fully cover round trip paths for each case study (section 3.2.2). Recall that the trees were generated using breadth first and depth first algorithms (section 3.2.1): the OrderedSet state machine led to seven unique trees using depth first algorithm and two trees using breadth first algorithm (in Appendix C); the CruiseControl state machine led to the same three trees regardless of the algorithm type being used because of its cyclic nature (Appendix D).

Table 6 shows the difference between the trees in terms of their size, as measured by the number of nodes, edges (with and without counting edges that appear several times), paths in the trees, their depth (i.e., the longest path from the root node to a leaf node, counting the number of edges in the path), and the number of round trip paths they fully cover. For full coverage, i.e., covering entirely round-trip paths and using prefixes, we also indicate the size that is due to those prefixes. For instance, for OrderedSet, the full coverage test suite covers 35 nodes: prefixes cover 8 nodes and round trip paths cover 27 nodes.
In order to guarantee a uniform measurement for the test suites regardless of their representation form or construction, and obtain comparable values, we represent the paths of the full coverage test suites as trees as shown in Appendix E sections E.3 and E.4. For example if we have two paths: (A)-x-(B)-y-(C) and (A)-x-(B)-z-(D), (A), (B), (C) and (D) being states and x, y and z being transitions, we do not count each path separately, which would lead to count six nodes and four edges. Instead consider the paths make up a tree as follows:

- The root node is A (if paths do not have the same initial state, we can always create a pseudo initial state);
- Node A has one outgoing edge labeled x, leading to node B;
- B has two outgoing edges labeled y and z, leading to C and D respectively.

Based on this transition tree, which has a structure similar to BFS/DFS trees, we can use the measure of size we defined for DFS/BFS trees, and then obtain four nodes and three edges.

Table 6 Characteristics of the generated transition trees

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>OrderedSet</th>
<th>BFS</th>
<th>Full coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DFS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trees</td>
<td>T1 T2 T3 T4 T5 T6 T7</td>
<td>T1 T2</td>
<td></td>
</tr>
<tr>
<td>Nodes</td>
<td>34 34 34 34 34 34 35 (8 in prefix)</td>
<td>34 34</td>
<td></td>
</tr>
<tr>
<td>Edges (no duplicates)</td>
<td>33 33 33 33 33 31</td>
<td>33 33</td>
<td></td>
</tr>
<tr>
<td>Edges (with duplicates)</td>
<td>97 112 80 106 113 57</td>
<td>57 52</td>
<td></td>
</tr>
<tr>
<td>Depth</td>
<td>8 9 6 8 9 3 3</td>
<td>3 3</td>
<td></td>
</tr>
<tr>
<td>Number of RTPs</td>
<td>19 20 19 18 20 19 18</td>
<td>17 21</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>BFS/DFS</th>
<th>Full Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trees</td>
<td>T1 T2 T3</td>
<td></td>
</tr>
<tr>
<td>Nodes</td>
<td>30 30 30 35 (7 in prefix)</td>
<td></td>
</tr>
<tr>
<td>Edges (no duplicates)</td>
<td>29 29 29 35 (6 in prefix)</td>
<td></td>
</tr>
<tr>
<td>Edges (with duplicates)</td>
<td>89 89 89 111 (82 in prefix)</td>
<td></td>
</tr>
<tr>
<td>Depth</td>
<td>5 5 5 5 5</td>
<td></td>
</tr>
<tr>
<td>Number of RTPs</td>
<td>23 23 23 23 23 23 23 23 23 23 23</td>
<td></td>
</tr>
</tbody>
</table>

54
From Table 6 we can see that the cost of fully covering RTPs fully is slightly higher than the cost of covering RTPs in pieces: the difference is negligible for OrderedSet, and a bit more significant for CruiseControl, especially when counting duplicated edges. One possible reason for this difference between the two systems is the coverage of RTPs by DFS/BFS trees: in the case of OrderedSet, the DFS/BFS trees cover on average 90.4% of the RTPs, and thus the cost needed to reach full coverage (100%) is slightly higher, while in the case of CruiseControl, the trees cover on average only 79.3% of the RTPs, thus the cost of reaching full coverage (100%) is sensibly higher.

Although DFS/BFS transition trees partially cover RTPs, they still cover a high percentage of those fully: 90.4% for OrderedSet, 79.3% for CruiseControl. We also notice that the algorithm that builds the tree strongly affects the depth of the tree (as expected): DFS trees are much deeper than BFS trees (two to three times) for OrderedSet. The algorithm also affects the number of Edges (with duplicates). In the case of OrderedSet, we observed that the DFS trees were repeating long prefixes, which result in high numbers of edges, while the BFS trees were repeating the shortest possible prefixes, which results in fewer numbers of edges.

For a given case study, the transition trees derived by a type of algorithm (i.e., either DFS or BFS) have very similar structures, measured by their numbers of nodes, edges, and paths. This is also true for trees built using different algorithms, except when counting all the edges they contain (with duplicates) in the case of OrderedSet: again this is due to the characteristics of the two state machines (CruiseControl’s state machine is strongly connected).
The variation observed in structures when looking at trees built by different algorithms is because the DFS algorithm tends to use the longest possible prefixes to cover round trip paths, while the BFS algorithm tends to use the shortest prefixes. The connectedness of the state machine (i.e., the extent to which every state is reachable from every other state) increases flexibility for the traversal algorithm: the higher the connectedness the higher the number of different prefixes of similar or equal sizes reaching round trips:

- For the OrderedSet case the difference between the longest and shortest possible prefixes to cover certain round trip is slightly high due to the weak connectedness. (For example we identified a round trip for which the longest prefix can have a length up to six while the length of the shortest possible prefix is 1). That is why we can observe slight variation in the structures using different algorithms for OreredSet.

- For CruiseControl the difference between the longest and shortest possible prefixes to cover certain round trip is very low due to the strong connectedness. (For many round trips, the length of the longest prefix equals the length of the shortest possible prefix.) In other words, the length of the prefix is not affected by or dependant on the algorithm type due to the strong connectedness nature of the state machine, and that is why the test suites derived using different algorithms have similar structures. In fact we have already discussed that DFS and BFS trees are the same. The slight cost increase in the full coverage suite is due to covering more round trips than DFS/BFS trees (29:23), which consequently requires more paths. Detailed description of these paths can be found in section 4.2
As pointed out in section 3.2.2, fully covering RTPs does not necessarily lead to covering all transition. This is the case for OrderedSet, for which the tree fully covering RTPs (and manually derived, using shortest prefixes) missed two transitions: see the transition tree in Appendix E section E.2; specifically:

- (Filled 2) -> add(n) -> (partially filled 3);
- (Filled 4) -> add(n) -> (partially filled 5).

From Table 6 we also note that prefixes that were required to fully cover RTPs, though shortest paths, amount for a substantial part of the total cost: 48%/20% (counting or not duplicates) of the edges and 22% of the nodes for OrderedSet, and 73%/17% (counting or not duplicates) of the edges and 20% of the nodes for CruiseControl.

4.2 Mutation score analysis for test suites

The mutation score for the test suite calculated in this section is the total number of unique faults detected by the path as a whole – the contribution of prefixes is discussed in section 4.3.4. Table 7 records the mutation scores of the BFS, DFS, and Full RTP coverage trees for the CruiseControl and OrderedSet case studies: the highest mutation score of DFS/BFS trees for CruiseControl is 50% (the full coverage tree scores 53%); the highest score of DFS trees for OrderedSet is 76% and the highest score of BFS trees for OrderedSet is 78% (the full coverage tree scores 66%).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>OrderedSet</th>
<th>BFS</th>
<th>Full coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trees</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mutation score</td>
<td>76%</td>
<td>74%</td>
<td>69%</td>
</tr>
<tr>
<td></td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
</tr>
<tr>
<td>CruiseControl</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Algorithm</td>
<td>BFS/DFS</td>
<td>Full Coverage</td>
<td></td>
</tr>
<tr>
<td>Trees</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mutation score</td>
<td>50%</td>
<td>45%</td>
<td>42%</td>
</tr>
<tr>
<td></td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
</tr>
</tbody>
</table>

Table 7 Mutation Scores for transition trees for CruiseControl and OrderedSet
The average score for the CruiseControl of all trees is 47.5% (45% when only counting BFS/DFS trees), which is considered low in general. This low score is mainly due to the real-time nature of the code, which is not really modeled in the state machine and therefore not really exercised by the trees. The “full coverage” test suite score is 53%, which is slightly better than the best BFS/DFS tree (treel) that achieved 50%. However, considering that “full coverage” suite has three more paths and 22 more edges (counting duplicates) than treel (Table 6) a higher mutation score was expected from “full coverage” test suite.

The average mutation score of DFS trees for OrderedSet is 73%, which is not as high as expected. However we observed that there are five or six paths (out of 24/25—Table 7) in each tree of length one and which have a very low fault detection capability (but have a cost).

To describe another reason for the low mutation score recorded by DFS trees of the OrderedSet, recall that the mutation score of the tree is the total number of the unique faults detected by the paths of the tree: for example if path1 detects 5 faults, and path2 detects 7 faults, 3 of them being already detected by path1, then the total mutation score of the two paths is not 5+7=12 but it is 5+ (7-3) =9. We noticed that although the paths of DFS trees have high scores, the scores are not perfect (they miss some faults) and paths are detecting approximately the same faults, and so the total mutation score of the tree is low.

These paths are detecting the same faults because they have the same prefix—see tree structures in Appendix C. This phenomenon can also be observed by looking at the mutation scores of test suites represented in Table 11 and paths scores in Table 13.
This can also be observed, though to a lesser extent for BFS trees and the tree covering RTPs fully. This suggests that (1) prefixes contribute a lot to the mutation score of trees, and (2) one would strive for varying prefixes to maximize the changes of detecting more faults.

These trees scored better than the "full coverage" test suite: the best tree scores 76% whereas "full coverage" scores 66%.

From Table 6 and Table 7 we also notice that:

1. Trees with the longest paths (e.g., trees 2, 6, and 7 for OrderedSet) did not achieve the best scores. Rather they recorded lower scores than other trees (e.g., tree 4) that have fewest and shortest paths. One exception is tree 3, but this tree has the smallest number of edges, i.e., it exercises the SUT the least. This phenomenon means that cumulative path length is not the only factor that can affect the mutation score, and that other factors have a negative impact on the score.

2. For OrderedSet, tree 4 scores much higher than tree 3 even though they have approximately the same structure (paths, edges, depth). This suggests that the exercised transition sequences have a significant impact on the score. The same phenomenon can be noticed in CruiseControl trees: The three trees have the same characteristics (nodes, edges, paths, depth) but varying scores.

3. Recall that the "full coverage" test suite of OrderedSet has no simple path: The 21 paths are all of type "Prefix + RTP". The low mutation score is due to: (i) The short length of paths compared to DFS/BFS trees (3 compared to 6/9 for DFS/BFS respectively); (ii) The smaller number of paths compared to DFS/BFS trees (21 compared to 24/25); (iii) The severe lack of simple paths compared to
DFS/BFS trees (0 compared to 6/7). The effect of the number of paths, paths length, and path type is precisely investigated in sections 4.2.1, 4.3.1 and 4.3.2 respectively.

4. Recall that the "full coverage" test suite of CruiseControl has no simple path, compared to two simple paths in DFS/BFS trees. This did not strongly affect the score negatively and a high score was still observed because the full coverage suite has (i) similar path lengths compared to DFS/BFS trees (5: 5) and (ii) higher numbers of paths compared to DFS/BFS trees (28: 25)

Table 7 illustrates that BFS trees for OrderedSet scored better than "full coverage" test suite and better than DFS trees, though with much smaller trees (Table 6). Again that emphasizes that the length (size) of paths, that we measure in terms of depth and number of edges, is not the only factor that affects the mutation score of a set of paths. We will investigate the quality of the paths and identify possible factors that affect path score in the coming sections.

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>StdDev</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>OrderedSet</td>
<td>66</td>
<td>78</td>
<td>73.5</td>
<td>3.74</td>
<td>10</td>
</tr>
<tr>
<td>CruiseControl</td>
<td>42</td>
<td>53</td>
<td>47.5</td>
<td>4.93</td>
<td>4</td>
</tr>
</tbody>
</table>

**Table 8 Mutation scores descriptive statistics of test suites**

Table 8 shows a statistical descriptive analysis that compares mutation scores for test suites for each case study. We can see that CruiseControl test suites had lower mutation scores compared to OrderedSet test suites. This could be due to the real-time behaviour of CruiseControl and the fact that this aspect is not really modeled in the state machine and therefore not tested on purpose. Indeed, some test cases were targeting time-dependant variables such as "car speed" and "throttle". The values of these variable changes constantly over time while the car is running, thus detecting the faults related to
these variables requires to stop and resume the Cruise control several times and/or execute the system for a long enough period of time which was not possible with the short test sequences. Another possible explanation is the high cyclic nature of its UML state machine that results in generating short paths (e.g., depth of trees equals four), and the existence of many self-transitions (e.g., 18 out of 28 that represents 64% of the total number of transitions). Indeed, in the case of CruiseControl, short paths tend to cover minimal portions of the code since parts of the code that are triggered after several break/accelerate commands or after a long enough period of time are not exercised. Similarly, self-transitions result in minimal number of states and transitions (1 event and 1 transition), which leads to low code coverage. The low standard deviation for both cases indicates that the variability of data dispersion points tend to be very close to the mean.

Table 9 shows a statistical descriptive analysis that compares the cost of trees and “full coverage” test suites in terms of number of edges (counting duplicates), paths, and nodes. The full coverage has higher cost in terms of number of nodes than the BFS/DFS trees, although the “full coverage” test suite may have a smaller number of paths as we can see in the OrderedSet case study. The higher cost of full coverage in the case of CruiseControl, measured in terms of number of edges is due to some long paths that are necessary to exercise specific round trips. These round trips are the result of having three outgoing and two incoming transitions between the Cruising state and the Standby state—check the state machine diagram in section 3.1.1. These transitions result in nine different round trips of length five (3 of the nine paths have prefix costs one and RTP costs four,
and the remaining 6 of the nine paths have prefix costs three and RTP costs two). Each of the DFS/BFS tree is covering three and missing six of these paths.

<table>
<thead>
<tr>
<th></th>
<th>OrderedSet</th>
<th></th>
<th>CruiseControl</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Trees</td>
<td>Full Coverage</td>
<td>Trees</td>
<td>Full Coverage</td>
</tr>
<tr>
<td>average number of paths</td>
<td>24.8</td>
<td>21</td>
<td>25</td>
<td>28</td>
</tr>
<tr>
<td>average number of nodes</td>
<td>34</td>
<td>35</td>
<td>30</td>
<td>35</td>
</tr>
<tr>
<td>average number of edges (with duplicates)</td>
<td>90.8</td>
<td>52</td>
<td>89</td>
<td>111</td>
</tr>
</tbody>
</table>

Table 9 Cost descriptive statistics of test suites

In order to verify whether covering more RTPs increases mutation score, we performed a comparative analysis between the mutation scores of the full coverage of RTPs and the partial coverage represented in BFS/DFS trees. We apply the one tailed t-test on the data of the total mutation scores of DFS/BFS transition trees and the “full coverage” presented in Table 7: we grouped the data of the two case studies together. Since applying the t-test requires checking the normal distribution of the data set, we used Shapiro-Wilk test for small samples, with high value of W=0.735 the data set is likely to be from normal distribution (using estimated parameter test with µ=67 , σ =13 the likelihood of fitness to the normal distribution is 95.16%). The research hypothesis of the t-test is that the sample mean of the partial round trip coverage is lower than the population mean of the full round trip coverage. Given that the average mutation score for the full coverage is 59.5, we set the null and alternative hypothesis as follows: Null hypothesis: $H_0$: $\mu \leq 59.5$ (the sample mean of partial coverage is lower than the population mean of full coverage); Alternative hypothesis: $H_a$: $\mu > 59.5$ (the sample mean of partial coverage is higher than the population mean of full coverage). We set the level of significance $\alpha = 0.05$. Table 10 shows the result of the t-test.
<table>
<thead>
<tr>
<th>Descriptive statistics</th>
<th>Sample Mean (µ)</th>
<th>StDev</th>
<th>Population Mean</th>
<th>n</th>
<th>DF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutation Score data</td>
<td>67.1667</td>
<td>13.2996</td>
<td>59.5 (hypothesized value)</td>
<td>12</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 10 Descriptive statistics for two sample t-test for test suites mutation scores

![Figure 10 one tailed t-test](image)

By looking in the t-distribution tables with DF\(^3\)=11 the tabled value for \(t_{0.5} = 1.79\). Since the calculated t-Value = 1.996 is greater than the tabled value of \(t_{0.5}\), the null hypothesis can be rejected. By considering the test statistic t value of Wilcoxon test = 1.7 we can still reject the null hypothesis. The mutation scores recorded by BFS/DFS trees are higher than the population average of full coverage trees. From this result we can conclude that covering more round trip paths is not an appropriate criterion to distinguish between test suites. The same analysis was done on each case study separately. The results are summarized in Appendix H: The same conclusion is drawn for OrderedSet, but not for CruiseControl, likely because of the small number of data points (three BFS/DFS trees for CruiseControl).

\(^3\) The number of degrees of freedom (DF) is the number of values of the independent parameters, which are in our context the number mutation scores -1.
Note that the small number of the data sample can be considered a threat to validity; however, this data sample is representing two common types of systems; a real-time reactive systems (CruiseControl) and data structure system (OrderedSet).

Based on the achieved results and observed facts we identified some factors that could affect the mutation score of a test suite such as: the number of paths, the number of transitions, the used algorithm, and the percentage of round trip path coverage. In the coming sections 4.2.1, 4.2.2, 4.2.3, and 4.2.4, respectively we tried to determine which of these factors have significant effect on the mutation score of the test suite.

We also investigate the impact of different factor on the mutation score of the path such as: The length of paths, the type of the path, the amount of self transitions, the prefix, the event sequences and the type of events in sections 4.3.1, 4.3.2, 4.3.3, 4.3.4, 4.3.5 and 4.3.6 respectively.

4.2.1 Studying the effect of the number of paths on the mutation score

We used the t-test for two unpaired samples to check the significance of the relationship between the mutation score of test suites and the number of paths. By using the information provided in Table 6 and Table 7 we calculate $t_{\text{OrderedSet}} = 40.8$ and $t_{\text{CruiseControl}} = 8.4$. Using the t-distribution tables the $P_{\text{OrderedSet}}$ value = 0.0001 and $P_{\text{CruiseControl}}$ value = 0.0002. This difference is considered highly statistically significant and the differences between numbers of paths strongly affect the mutation score of the test suites for each case study. Using the correlation coefficient method discussed in section 3.6 the coefficient $r$ for OrderedSet = 0.68 and for CruiseControl = 0.749. The value of $r$ indicates the positive effect of the number of paths on the mutation score. Figure 11 illustrates the relation.
4.2.2 Studying the effect of the number of transitions on the mutation score

We used the t-test for two unpaired samples to check the significance of the relationship between the mutation score of test suites and the number of edges (with duplicates). By using the information provided in Table 6 and Table 7 we calculate $t_{\text{OrderedSet}} = 8.13$ and $t_{\text{Cruise Control}} = 14.4$. Using the t-distribution tables the $P_{\text{OrderedSet}}$ value = 0.0001 and $P_{\text{Cruise Control}}$ value = 0.0007. This difference is considered statistically significant and the differences between the numbers of transitions strongly affect the mutation score of the test suites in the case of CruiseControl and slightly affect the score in the case of OrderedSet, but both effects are positive. Using the correlation coefficient method discussed in section 3.6 the coefficient $r$ for OrderedSet = 0.743 and for CruiseControl = 0.023. The value of $r$ indicates the positive effect of the number of paths on the mutation score. Figure 12 illustrates the relation.

Figure 11 Correlation between number of paths and mutation scores
Figure 12 Correlation between number of transitions and mutation scores

4.2.3 Studying the effect of different algorithms on the mutation score

We used the t-test for two unpaired samples to check the significance of the relation between mutation scores of transition trees generated using a depth first algorithm and a breadth first algorithm to identify the effect of the algorithm on the score. We are trying to know whether there is a significant (or only random) difference in the average mean between mutation scores of DFS trees and BFS trees.

The null hypothesis assumes the average difference between the two population means of DFS trees and BFS trees are equal, and the alternative hypothesis assumes that the average difference between the two population means is really significant.

By using the information provided in Table 6 and Table 7 we calculate \( t = 1.95 \) and \( p \) value = 0.1223. This difference is considered to be not statistically significant and we cannot reject the null hypothesis, thus the algorithm types do not significantly affect the mutation score of the transition tree.

The same study was performed on OrderedSet only to remove the (possible) noise caused by the CruiseControl since the CruiseControl has identical BFS and DFS trees.
The $t_{\text{OrderedSet}} = 2.8$ and $P_{\text{OrderedSet}}$ value = 0.024 which is considered statistically significant, and we can reject the null hypothesis.

To help visualize the results, we used mean difference plot as shown in Figure 13.

![Mean Difference Plot](image)

**Figure 13 mean difference plot for scores of trees generated by different algorithms**

As we can see from Figure 13 all differences are positive, which means that BFS trees have higher scores than DFS trees, the average pair difference line indicates that BFS trees have a score 1.2 times higher than the DFS trees. With 95% confidence we can say that BFS trees can have a score at the maximum 3.4 higher than DFS trees and a score at the minimum -1 lower than DFS trees.

Because we have a small sample data set we also applied Wilcoxon test for non parametric values. The statistic $t$ value = 1.5 is still close to the $t$-test value and confirms the insignificant effect of the algorithm on the mutation score.

It is also worth to point out another conclusion that was derived from statistical analysis but that we do not report about. There is no significant effect of the algorithm type on the number of paths.
4.2.4 Studying the effect of round trip coverage percentage on mutation score

Although DFS/BFS transition trees cover a high percentage of round trip paths, they do not include all possible round trip paths [11]. Table 11 shows the percentage of round trip paths covered in each test suite. The total number of possible round trip paths that could be extracted from the UML state machine is 29 for CruiseControl and 21 for OrderedSet. All CruiseControl transition trees cover 23 (79%) of the 29 RTPs. OrderedSet DFS trees cover 18 to 20 of the 21 RTPs (85.7% to 95.2%) and BFS trees cover 17 or 18 of them (80% or 85%). Each BFS/DFS tree is missing different round trip paths, however all the round trip paths are covered in one or more of the trees (for each case study), and there is no round trip path that is not covered by any DFS/BFS tree.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>OrderedSet</th>
<th></th>
<th>BFS</th>
<th>Full Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trees</td>
<td>DFS</td>
<td>T1 T2 T3 T4 T5 T6 T7</td>
<td>T1 T2</td>
<td></td>
</tr>
<tr>
<td>Tree Mutation Score</td>
<td>76% 74% 69% 75% 73% 73% 73%</td>
<td>78% 78%</td>
<td>66%</td>
<td></td>
</tr>
<tr>
<td>% of RTPs covered</td>
<td>76% 72% 76% 76% 76% 76% 76%</td>
<td>85% 80%</td>
<td>100%</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>DFS/BFS</th>
<th>Full coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trees</td>
<td>T1 T2 T3</td>
<td></td>
</tr>
<tr>
<td>Tree Mutation Score</td>
<td>50% 45% 42%</td>
<td>53%</td>
</tr>
<tr>
<td>% of RTPs covered</td>
<td>79% 79% 79%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 11 RTP coverage in transition trees (percentages)

We used a statistical correlation method—discussed in section 3.6—to find the degree of relationship between the two variables: the percentage of total RTPs covered and the mutation score of the test suites. To insure the analysis is not biased by the number of paths of the each test suites we excluded the full coverage suite for the CruiseControl as it has 6 paths more than DFS/BFS trees, and the full coverage suite for the OrderedSet as it has 4 paths less than DFS/BFS trees (see Table 6).

We used the data from Table 11 to calculate the statistical correlation r between the percentage of RTPs covered and the mutation score of the test suite. The correlation
value $r$ for OrderedSet is 0.6 and indicates a positive relationship between the two variables. Given a correlation coefficient of 0.6 for a sample size of 9, the 95% Confidence Interval on the Population Correlation is from -0.11 to 0.91. Figure 14 shows that the higher the coverage of round trip path in a test suite the higher the mutation score.

![Mutationscore vs. Percentage of RTPs covered](image)

**Figure 14 Scattered diagram showing positive correlation between RT coverage and mutation score (OrderedSet)**

### 4.3 Mutation score analysis for test cases (path level)

In this section we try to identify the factors that can affect the mutation score of the path such as path length, path type, inclusion of self-transitions, prefix, and event sequence.

#### 4.3.1 Impact of complete path length (Size) on the mutation score

Table 12 shows the size effect on the mutation score of each complete path on all test suites. A clear direct correlation between path length and the mutation can be found by analysing the scores in Table 12. The mutation score increases significantly with every path unit increase regardless of the path type. This is an expected result since when we increase length we do more testing and therefore increase fault detection.
Table 12 Mutation Scores for paths according to their length (CruiseControl)

From Table 12 we can see that the longer the path length (i.e., number of edges) the better the score, even though some long paths scored lower or the same as shorter paths: length five paths in tree 2 have a minimum score of 79, which is the maximum score for length four paths; in tree 3 some length four paths scored much better than length five paths, 75 and 58 respectively. Another observation is that paths of a given length are not equal with respect to mutation score, either within a tree or among trees: e.g., paths of length two in tree 3 score between 6 and 25; paths of length two in DFS/BFS trees score on average between 22 and 25.

The variation of scores between DFS/BFS paths is less than the variation of scores of full coverage paths, because the paths of the tree structure share the same prefix (section 4.2), and thus tend to catch a common (sub) set of faults.

The variation of score within a test suite is due to: (1) the number of paths of identical length (for example the full coverage has 4 paths of length four, but 12 paths of length five, thus there is high variation of scores of length five paths); (2) the length of the shared prefix between paths has the same length (for example the paths of length five in tree 1 share the same prefix of length four, thus there is low variation of scores).

The variation of scores within paths of identical length is caused by different factors such as the path type, the sequence of events in the path, the percentage of the prefix length to the round trip length: e.g., path of length three can be cost two for the prefix and
cost one for the RTP or cost one for the prefix and two for the RTP, or cost zero for the prefix and three for the RTP, or cost three for the prefix and zero for the RTP (recall that these are paths in BFS/DFS trees, and some of them may not cover any RTP fully). The impact of each of these factors will be investigated in coming sections.

<table>
<thead>
<tr>
<th>Length 1</th>
<th>Length 2</th>
<th>Length 3</th>
<th>Length 4</th>
<th>Length 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>Avg</td>
<td>Range</td>
<td>Avg</td>
<td>Range</td>
</tr>
<tr>
<td>2 BFS trees</td>
<td>[6-12]</td>
<td>11%</td>
<td>[28-48]</td>
<td>34.5%</td>
</tr>
<tr>
<td>Full Cover</td>
<td>-</td>
<td>-</td>
<td>[16-45]</td>
<td>25%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Length 6</th>
<th>Length 7</th>
<th>Length 8</th>
<th>Length 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>Avg</td>
<td>Range</td>
<td>Avg</td>
</tr>
<tr>
<td>7 DFS trees</td>
<td>[45-80]</td>
<td>74%</td>
<td>[56-72]</td>
</tr>
<tr>
<td>2 BFS trees</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Full Cover</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table 13 Average mutation scores for paths according to their length (OrderedSet)**

For the OrderedSet case study because of the large number of trees, an average of mutation score was calculated in Table 13. We can see that the score increases with the size increase until length four for the DFS trees where the score starts decreasing dramatically to the extent that length eight paths score less than length four paths. This dramatic decrease in the score will be investigated in the coming sections. Figure 15 presents the frequency distribution of path lengths across all test suites for both case studies. The OrderedSet test suites cover all path lengths from one to nine. The maximum length of paths for the CruiseControl is five.
The majority of the generated paths are of length three and the frequency is minimum for length nine. Generating many short length paths does not add much to the mutation score but rather appears a worthless addition to the cost as shown in Table 12 and Table 13. This observable fact is considered a weakness of the existing algorithms to cover round trip paths that we tried to avoid in the new proposed algorithm discussed in Chapter 5.

We used a statistical correlation method (section 3.6) to find the degree of relationship between two variables: the variable representing the path length and the variable representing average mutation score for paths of the same length. The correlation $r = 0.79$ indicates a fairly strong positive relationship between the two variables. The relation is presented in a scattered diagram in Figure 16.
To get more evidence on the effect of length on the mutation score we build two random sample test suites for each case study: Appendix F describes the path sequences of the random test suites. The random test suites have the same structures (number of paths and length of paths) as the “Full Coverage” test suites: The OrderedSet random test suite contains 22 paths: eleven of length two, and ten of length three; The CruiseControl random suite contains 29 paths: six of length two, six of length three, four of length four, and 13 of length five. When choosing the random paths we tried to cover the maximum possible variations of states, transitions and events and avoid repeating the same sequences in an attempt to achieve better code coverage. Therefore the random test suites are not purely random, which may make results biased to random testing by making the suites more effective than what random may produce.

The scores of the random test suites and the “Full Coverage” test suites are compared in Table 14. The scores of the random test suites are higher than “Full Coverage” suite in the OrderedSet case, and very competitive in the CruiseControl case. From this result we can deduce that: the used pattern of number and length of paths enabled the random suite of the CruiseControl to maintain relatively high score, which confirms the positive effect of path length on the path score, hence the mutation score of the test suite. For the OrderedSet the random suite was even more effective than full coverage, and since we used the same pattern of number and length of paths, this higher score may be due to the construction technique used that aimed to achieve higher variability of states and events.

<table>
<thead>
<tr>
<th></th>
<th>CruiseControl</th>
<th>OrderedSet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Coverage test suite</td>
<td>53%</td>
<td>66%</td>
</tr>
<tr>
<td>Random test suite</td>
<td>50%</td>
<td>74%</td>
</tr>
</tbody>
</table>

Table 14 Random test score compared to all round trip score
We checked the variability in terms of number of repeated states and events per path for each case study. For the CruiseControl, the full coverage tree has 29 paths that include one repeated state (as opposed to 26 for random suite), and eight paths that include one repeated event (as opposed to four for random suite). For the OrderedSet, the full coverage tree has 22 paths that include one repeated state (as opposed to 14 for random suite), but none of the paths includes any repeated event (as opposed to one for random suite). From this analysis we can deduce that random test suites / trees have lower variability in terms of states and events per path for both case studies. We also see that full coverage trees have few repeated states and events, suggesting higher variability.

### 4.3.2 Impact of path type on the mutation score

When studying the path type we have to consider the size factor at the same time. We have to compare mutation scores of different path types of the same length. As shown in Table 15 only paths of length four cover both types of paths across all test suites of CruiseControl.

<table>
<thead>
<tr>
<th>Path type</th>
<th>Tree1</th>
<th>Tree2</th>
<th>Tree3</th>
<th>Full Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
<td>Min</td>
<td>Max</td>
</tr>
<tr>
<td>Prefix+RTP</td>
<td>45</td>
<td>48</td>
<td>43</td>
<td>45</td>
</tr>
<tr>
<td>Simple</td>
<td>62</td>
<td>81</td>
<td>57</td>
<td>79</td>
</tr>
</tbody>
</table>

*Table 15 Mutation Scores for paths according to path type with respect to length (CruiseControl)*

By looking at Table 15 we can notice that for paths of length four, the simple paths always have higher minimum and maximum scores than the Prefix+RTP paths. We believe this is due to the sequence of transitions in the prefix, the prefix length, and the percentage of the prefix over to the whole path length. More investigation on the prefix effect is done in section 4.3.4.
A dependent t-test for unpaired samples (section 3.6) was performed to study the impact of path type on the mutation score of the path: Table 16. The results show that there is a significant difference between Simple paths and the “Prefix + RTP” paths.

<table>
<thead>
<tr>
<th>Type of Path (of Length 3)</th>
<th>DF</th>
<th>Mean of difference in mutation score</th>
<th>t value</th>
<th>p value</th>
<th>Statistical significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CruiseControl</td>
<td>Simple vs. Pre+RTP</td>
<td>19</td>
<td>25.43</td>
<td>10.06</td>
<td>0.0001</td>
</tr>
<tr>
<td>Order Set</td>
<td>Simple vs. Pre+RTP</td>
<td>68</td>
<td>5.72</td>
<td>2.30</td>
<td>0.0245</td>
</tr>
</tbody>
</table>

Table 16 Impact of path type on the mutation score - Unpaired t-tests results

For OrderedSet the analysis is limited because:

- The “Full coverage” test suite includes “Prefix + RTP” paths of lengths two and three only;
- The simple paths of the DFS trees are all of length one, which does not allow comparisons with the “Prefix + RTP” paths which require at least length two.

Fortunately, BFS trees have simple paths of length two and three.

However, from Table 17 we can see that simple paths of BFS trees maintain higher minimum score than similar paths in the full coverage and DFS trees suite for both length two and three, but do not always achieve the maximum score (e.g., maximum score length three paths of BFS trees).

The t-test statistics results confirms the significance of the mean difference between mutation scores of simple and “prefix + RTP” paths as calculated in Table 16.
### Table 17 Mutation scores for paths according to path type with respect to length (OrderedSet)

We can conclude that, at equal length, the mutation score of simple paths is higher than the mutation score of "Prefix+RTP" (full coverage) paths. The variability of states could be one of the reasons of the high scores achieved by Simple paths: since simple paths do not contain RTPs, they traverse more states than Prefix+RTP paths of identical length.

#### 4.3.3 Impact of self-transitions on the mutation score

The analysis of self-transitions effect will help us explain the variation of mutation score of paths that are of the same length and type. To check the effect of self-transitions we compare paths of identical size and type. Note that there is only one type of paths, "Prefix+RTP" paths, that include self-transitions because self-transitions are by definition round trips. Since simple paths do not include round trips, the selected paths for this analysis (Table 18 and Table 19) are of type "Prefix+RTP". Therefore we distinguish between the paths that do not include self-transition ("Prefix+NoSelfTrans") and the paths that include self-transitions ("Prefix+SelfTrans"). Note also that due to the way DFS/BFS trees are constructed (actually the termination criterion), paths that include self-transitions can only contain the self-transition at the end of the path and are therefore composed of a prefix and a self-transition. The tables show the average mutation scores.
of paths extracted from all test suites of each case study. We observe that paths that include a self-transition scored less than paths of identical length and type that do not, which means that hitting the same state successively with self-transitions, has negative impact on the path score.

<table>
<thead>
<tr>
<th>Path type</th>
<th>Path Length</th>
<th>Number of Paths</th>
<th>Min</th>
<th>Max</th>
<th>Stdev</th>
<th>Average Mutation score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefix+NoSelfTrans</td>
<td>5</td>
<td>15</td>
<td>64%</td>
<td>95%</td>
<td>12.6</td>
<td>82%</td>
</tr>
<tr>
<td>Prefix+SelfTrans</td>
<td>20</td>
<td>58%</td>
<td>73%</td>
<td>9.1</td>
<td>60%</td>
<td></td>
</tr>
</tbody>
</table>

Table 18 Impact of self-transition on mutation score (CruiseControl)

<table>
<thead>
<tr>
<th>Path type</th>
<th>Path Length</th>
<th>Number of Paths</th>
<th>Min</th>
<th>Max</th>
<th>Stdev</th>
<th>Average mutation score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefix+NoSelfTrans</td>
<td>3</td>
<td>21</td>
<td>37%</td>
<td>57%</td>
<td>8.4</td>
<td>44%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>11</td>
<td>35%</td>
<td>63%</td>
<td>12.3</td>
<td>68%</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>2</td>
<td>60%</td>
<td>64%</td>
<td>2.8</td>
<td>62%</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>6</td>
<td>58%</td>
<td>74%</td>
<td>6.1</td>
<td>71%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>5</td>
<td>44%</td>
<td>67%</td>
<td>9.0</td>
<td>59%</td>
</tr>
<tr>
<td>Prefix+SelfTrans</td>
<td>3</td>
<td>40</td>
<td>35%</td>
<td>48%</td>
<td>7.2</td>
<td>38%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>24</td>
<td>29%</td>
<td>58%</td>
<td>11.6</td>
<td>56%</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>14</td>
<td>49%</td>
<td>56%</td>
<td>2.6</td>
<td>55%</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>14</td>
<td>43%</td>
<td>68%</td>
<td>8.7</td>
<td>61%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>10</td>
<td>35%</td>
<td>59%</td>
<td>9.2</td>
<td>47%</td>
</tr>
</tbody>
</table>

Table 19 Impact of self-transition on mutation score (OrderedSet)

We used the t-test for two unpaired samples to identify the effect of including self-transitions on the mutation score of the path. The first data set is the mutation scores of paths that include a self-transition, and the second data set is the mutation scores of paths that do not include a self-transition. The calculated t-value = 1.57, and the mean difference = 8.71. This difference is not statistically significant. Thus, inclusion of self-transitions has no significant impact on the mutation score of the path. (Identical results were achieved when studying length 3, 4 together and length 5, 6, 8 together and the mean difference is not statistically significant.)
4.3.4 Impact of prefix on the mutation score of the path covering RTPs

To study the effect of the prefix of paths covering RTPs on mutation score we analyse the percentage of mutants killed by the prefix portion only and the percentage of mutants killed by the round trip portion only. We further perform the analysis according to the paths’ lengths and the contribution of the prefix and RTP to the paths’ length.

Table 20 shows that for the CruiseControl, paths covering RTPs have length two, three, four or five and that in paths of length greater than two, the prefixes have varying contribution to the length of the path: paths of length two all have a prefix of length one and a RTP of length one (i.e., 50% (1:1) in column “Prefix portion”); one third of the paths of length three have a prefix of length one and a RTP of length two (i.e., 33% (1:2)).

The set of mutants killed by the path is composed of (1) the mutants killed only by the prefix, (2) the mutants killed only by the RTP, and (3) the mutants killed by both the prefix and the RTP. In Table 20 we do not report on the mutants killed by both the prefix and the RTP. We only focus on those killed either by the prefix only or by the RTP only. The table indicates for instance that for paths of length 2, on average, 50% of the mutants killed by the path are killed by the prefix only (not the RTP) and 31% of the mutants killed by the path are killed by the RTP only (not the prefix). In other words, on average, the prefix misses 31% (at least, since we do not report here on mutants missed by both the prefix and the RTP) of the mutants killed by the path and the RTP misses 50% (at least) of the mutants killed by the path.
From Table 20 we can see that the prefix has in general a much higher contribution to the mutation score than the RTP portion. There is one exception: the four paths of length four; the difference is due to the sequence of events in those paths, which we discuss in section 4.3.5. The effect of length is obvious for both prefix and RTP: the longer the higher the contribution to the total mutation score of the path. The average mutation score of prefixes of length one, two, three and four is 39%, 58%, 61%, and 70%, respectively.

The prefix contribution is affected by the contribution of the RTP and how many mutants are killed by both RTP and prefix. This explains why we can notice a variation of contribution between prefixes of the same length: ex. prefix portion 25% (length 4) scored 12% and prefix portion 20% (length 5) scored 40%.

In addition, when studying the cumulative mutation score of the prefix and the RTP parts one transition at a time, the score of the prefix increases faster than the score of the RTP. This can be noticed from the scattered diagram in Figure 17 (a) that shows the effect of unit increase on the mutation score of the prefix and the RTP portions.
Figure 17 Scattered diagrams showing the effect of unit increase of the prefix and RTP portion on the mutation score of the path

The same analysis was performed on OrderedSet. The results are shown in Table 21 and Figure 17 (b). We can again see higher contributions of the prefix to mutation score than the RTP portion. The high contribution of the prefix was expected as we find from our previous analysis that simple paths are generally scoring better than round trip paths.

<table>
<thead>
<tr>
<th>Path type</th>
<th>Length</th>
<th>Number of paths</th>
<th>Prefix portion</th>
<th>Average mutation scores of prefix</th>
<th>Average mutation score of RTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefix</td>
<td>2</td>
<td>16</td>
<td>50% (1:1)</td>
<td>65%</td>
<td>28%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9</td>
<td>33% (1:2)</td>
<td>50%</td>
<td>15%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>28</td>
<td>66% (2:1)</td>
<td>68%</td>
<td>20%</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>6</td>
<td>25% (1:3)</td>
<td>30%</td>
<td>40%</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>5</td>
<td>50% (2:2)</td>
<td>45%</td>
<td>31%</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>1</td>
<td>75% (3:1)</td>
<td>70%</td>
<td>20%</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>14</td>
<td>80% (4:1)</td>
<td>80%</td>
<td>9%</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1</td>
<td>60% (3:2)</td>
<td>65%</td>
<td>12%</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>4</td>
<td>83% (5:1)</td>
<td>77%</td>
<td>15%</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>10</td>
<td>85% (6:1)</td>
<td>82%</td>
<td>10%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>5</td>
<td>75% (6:2)</td>
<td>65%</td>
<td>19%</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>8</td>
<td>87% (7:1)</td>
<td>90%</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>8</td>
<td>88% (8:1)</td>
<td>84%</td>
<td>7%</td>
</tr>
</tbody>
</table>

Table 21 Prefix and RTP scores according to their length (OrderedSet)
As we can understand the effect of the prefix on the path score, we can realize the
effect of the prefix on the test suite score. The shortest possible prefix were used when
we build full coverage test suites (maximum prefix length is 2); however for transition
trees the sequence of paths does not always use the shortest possible prefix and that could
be a reason for the higher mutation scores reached by those trees. The average mutation
score for short prefixes of full coverage suite is 35% for length 1 and 58% for length 2.
The average mutation score for long prefixes of the DFS/BFS trees is 67% for length 3,
80% for length 4, 77% for length 5 and 82% for length 6.

As shown from Table 20, short prefixes (of length 1 and 2) have better contribution to
the total mutation score of the path, than short round trips (of length 1 and 2). For
instance: prefixes of length 4 score 70% while RTPs of length 4 score 35%; prefixes of
length 2 score 58% while RTPs of length 2 score 26% and 30%. Since the total score of
the “Prefix+RTP” depends on the prefix and the round trip, we should consider giving
better chance for the prefix to score, especially if the round trip path that we intend to
cover is small. By choosing the longest prefix for the small round trip—even if no prefix
is needed—we can increase the effectiveness of the path. This will also overcome the
weaknesses of small round trip paths (self-transitions) and ensure a balance of
effectiveness between the prefix and the RTP. In addition, using long prefix for short
round trips will ensure a high minimum length for the path, as we proved previously the
strong positive relation between path length and score.

4.3.5 Impact of events on the mutation score

In this section we discuss the variation of mutation score between different events,
and the variation of mutation scores between same events. Events can access different
sections of the code, and different amounts of code lines. Since there is a high variation in the mutant distribution across code lines (functions and classes) in each case study, as shown in Figure 8 and Figure 9, the events have varying probabilities of detecting faults and thus different contributions to the path score.

For the CruiseControl case study, the events that access more than one class have a higher probability of detecting faults than other events (e.g., events engineOn, engineOff, accelerator, and brake access, i.e., call methods in, classes CruiseControl and Controller).

Table 22 presents paths of length 2 with different events. We can notice that paths with event "accelerator" have higher scores than other paths. We choose the length 2 paths from all test suites because they share the same first event, specifically the constructor event, thus we can observe the contribution of the second event.

From the table we can notice that the three highest scores (27%, 25%, and 19%) are for events that access three classes rather than two classes for other events (note that CarSimulator class has the highest mutant distribution). The amount of code accessed has no obvious effect on the mutation score because mutants are not uniformly distributed in the code.
We also noticed a variation of mutation score for events that have parameters (e.g., add(...)). For the OrderedSet case study, the events call varying numbers of functions depending on the start state, the type of input value (e.g., new or existing value), and satisfied guard conditions. For example, if we are calling event add() with a new value n and the current state is partially filled, then four functions are called as shown in Table 23. But if the current state is filled, the array will need to be resized and an additional resizearray() function is called (only if the set size is not exceeding the maximum allowed value and the number of resizes is not exceeding the maximum allowed value).

<table>
<thead>
<tr>
<th>event</th>
<th>start state (Current)</th>
<th>input value type</th>
<th>add()</th>
<th>functions accessed</th>
<th>LOC</th>
<th>classes accessed</th>
</tr>
</thead>
<tbody>
<tr>
<td>accelerator</td>
<td></td>
<td></td>
<td></td>
<td>handelcommand()</td>
<td>accelerate()</td>
<td>22</td>
</tr>
<tr>
<td>break</td>
<td></td>
<td></td>
<td></td>
<td>handelcommand()</td>
<td>public break()</td>
<td>20</td>
</tr>
<tr>
<td>off</td>
<td></td>
<td></td>
<td></td>
<td>handelcommand()</td>
<td>off()</td>
<td>8</td>
</tr>
<tr>
<td>resume</td>
<td></td>
<td></td>
<td></td>
<td>handelcommand()</td>
<td>resume()</td>
<td>8</td>
</tr>
<tr>
<td>engine off</td>
<td></td>
<td></td>
<td></td>
<td>handelcommand()</td>
<td>Publicengineoff()</td>
<td>16</td>
</tr>
<tr>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td>handelcommand()</td>
<td>on()</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 23 event code coverage (OrderedSet)

To prove the effect of the starting state on the mutation score of an event, we track the contribution of event add() to the path score with different starting states for the add() event in the following path (test case):
(Initial Node) -> OrdSet(int[1]v) -> (Partially Filled 1) -> add(n) -> (Filled 2) -> add(n) -> (Partially Filled 3) -> remove(x) ->(Partially Filled 3).

This path/test case was able to catch 223 mutants. The first add(n) event caught 58 unique mutants (26% of the path score). The second add(n) event caught 73 unique mutants (32% of the path score). This difference is due to accessing additional function "resizearray()" that has a moderate distribution of mutants as shown in Figure 9.

Similar observations were made for CruiseControl, there is high contribution of the unique sequence [(cruising)-> enginoff -> (Idle)] that exists only in one path of length four in every tree to the score of the path. In other words, this path is so specific that it is required to kill specific mutants, and no other path can do that (at least in the trees we analyze).

4.3.6 Impact of the repeated use of events on mutation score

In order to study the effect of using a certain event frequently in a single path we have to fix other variables (factors) that affect the mutation score of a path such as path length, path type and sequence of events of the path.

There are seven events in CruiseControl, and eight in OrderedSet: Table 24. The number of occurrences of each event in the state machine is also reported in the table.

Recall from section 3.1.2 that there are three input values that distinguish add and remove events for the OrderedSet: x, n, and val; where x is an integer, n is a new integer that is not included in the set, and val is an integer that is already included in the set.
To analyze the effect of event repetition in the case of CruiseControl, we extracted paths with the most repetitions across all (BFS/DFS and full coverage) suites and classified them into groups according to the number of occurrences of the repeated events: Table 25. For example for path (1, 8, 15, 21, 29) where transitions 8 and 29 have the same event, i.e., EngineOn (transition numbers underlined), the number of occurrences of the repeated event is 1 (it is repeated once) and the path score is 80%.

From Table 25 we can see that most of the paths that have at least one event repeated once scored less than other paths of the same length: for instance, sequence (1, 8, 15, 21). The average score of paths with one repeated event is 80% whereas the average score for paths with no repeated events is 85%. We performed a t-test to know whether there is a significant difference in the average scores between paths with no repeated events and paths with one repeated event. The null hypothesis assumes the averages between the two groups are equal. The result of the t-test shows that this difference is considered to be not statistically significant \( t=1.3 \) and \( p \text{ value}=0.19 \) and thus we can not reject the null hypothesis. We also noticed that the least score is recorded when the same event was

<table>
<thead>
<tr>
<th>CruiseControl</th>
<th>Event</th>
<th>Transition number</th>
<th>Occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>On</td>
<td>7,15,18,24</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Off</td>
<td>4,13,20,27</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>EngineOn</td>
<td>8,14,19,29</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>EngineOff</td>
<td>6,9,16,23</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>Accelerator</td>
<td>2,12,21,28</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>Break</td>
<td>3,11,22,26</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>Resume</td>
<td>5,10,17,25</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OrderedSet</th>
<th>Event</th>
<th>Transition number</th>
<th>Occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OrdSet(int[v])</td>
<td>1,2,4,5,6,7</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>OrdSet(x)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Remove(x)</td>
<td>8,18,25,32</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>Remove(val)</td>
<td>10,16,23,30</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>Remove(n)</td>
<td>11,14,21,28</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>Add(x)</td>
<td>9,33</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>Add(val)</td>
<td>12,15,19,22,26,29</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>Add(n)</td>
<td>13,17,20,24,27,31</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 24 Classification of transitions according to their event
used consecutively: highlighted paths. Although we only observed single repetitions of events, we conjecture that the effect of repeating events on score is inversely proportional to the amount of repetition, although the extent of the impact on score varies from one path to another.

The same study could not be performed on OrderedSet due to limited sample data: the longest sequence with repetition across the trees is (2,9,13), and there is no repeated event in that sequence, and so the comparison is not possible.

<table>
<thead>
<tr>
<th>Path Type</th>
<th>Path Sequence</th>
<th>Path Length</th>
<th>Path Score</th>
<th>Occurrence of repeated event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence 1, 8, 15, 21</td>
<td>5</td>
<td>P10: 1, 8, 15, 21, 29</td>
<td>80%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P5: 1, 8, 15, 21, 24</td>
<td>82%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P9: 1, 8, 15, 21, 28</td>
<td>78%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P4: 1, 8, 15, 21, 23</td>
<td>82%</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P6: 1, 8, 15, 21, 26</td>
<td>90%</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P7: 1, 8, 15, 21, 25</td>
<td>88%</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P8: 1, 8, 15, 21, 27</td>
<td>80%</td>
<td>0</td>
</tr>
<tr>
<td>CruiseControl</td>
<td>Prefix +RTP</td>
<td>Sequence 1, 8, 15, 22</td>
<td>5</td>
<td>P10: 1, 8, 15, 22, 29</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P5: 1, 8, 15, 22, 24</td>
<td>78%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P6: 1, 8, 15, 22, 26</td>
<td>75%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P4: 1, 8, 15, 22, 23</td>
<td>78%</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P7: 1, 8, 15, 22, 25</td>
<td>82%</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P8: 1, 8, 15, 22, 27</td>
<td>78%</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P9: 1, 8, 15, 22, 28</td>
<td>90%</td>
<td>0</td>
</tr>
<tr>
<td>Sequence 1, 8, 15, 21</td>
<td>5</td>
<td>P12: 1, 8, 15, 20, 29</td>
<td>72%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P7: 1, 8, 15, 20, 24</td>
<td>71%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P10: 1, 8, 15, 20, 27</td>
<td>66%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P6: 1, 8, 15, 20, 23</td>
<td>73%</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P8: 1, 8, 15, 20, 26</td>
<td>70%</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P9: 1, 8, 15, 20, 25</td>
<td>67%</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P11: 1, 8, 15, 20, 28</td>
<td>78%</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 25 Effect of event frequency on path score (CruiseControl)

4.4 Fault detection across different fault types

This section analyzes, for different mutation operators, the percentage of mutants killed by DFS/BFS trees and full coverage suites. Table 26 presents the fault detection effectiveness, across the different mutant operators presented in section 3.3. The notation "compile" is used to indicate mutants that were detected by compiler.
In both case studies a higher percentage of traditional mutants were killed than class mutants. For the OrderedSet case study the exact same types of mutants were detected either by the BFS/DFS trees or the full coverage. In general BFS/DFS trees have higher detection scores for all mutants except ROR, AODU, and COD, where the trees have equal scores to the full coverage. For the CruiseControl case study, some mutants were detected only by BFS/DFS trees (e.g., JID) and others were detected only by full coverage (e.g., EAM). In order to identify the reasons for not detecting specific types of faults in CruiseControl we traced the execution of test drivers against these mutants and watched variable values using eclipse debugging tools.

The following are examples of mutants that could not be killed by some or all of the test suites (see Appendix I):

- JSI (static modifier insertion): The “static” modifier keyword change was not detected because a static variable is not attached to a particular object, but rather to the class as a whole, and for the CruiseControl test driver there is only one instance of the system running. But still the class object can access these static

<table>
<thead>
<tr>
<th>Class</th>
<th>Mutation Operator</th>
<th>CruiseControl</th>
<th>OrderedSet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Trees</td>
<td>Full Cover</td>
</tr>
<tr>
<td>JID</td>
<td>22%</td>
<td>0%</td>
<td>-</td>
</tr>
<tr>
<td>JSD</td>
<td>Compile</td>
<td>9%</td>
<td>0%</td>
</tr>
<tr>
<td>JSI</td>
<td>0%</td>
<td>20%</td>
<td>0%</td>
</tr>
<tr>
<td>EAM</td>
<td>0%</td>
<td>60%</td>
<td>75%</td>
</tr>
<tr>
<td>JDC</td>
<td>0%</td>
<td>60%</td>
<td>75%</td>
</tr>
<tr>
<td>AOIS</td>
<td>52%</td>
<td>75%</td>
<td>81%</td>
</tr>
<tr>
<td>AOIU</td>
<td>50%</td>
<td>48%</td>
<td>81%</td>
</tr>
<tr>
<td>AORB</td>
<td>13%</td>
<td>13%</td>
<td>80%</td>
</tr>
<tr>
<td>ASRS</td>
<td>67%</td>
<td>67%</td>
<td>-</td>
</tr>
<tr>
<td>COI</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>LOI</td>
<td>64%</td>
<td>67%</td>
<td>87%</td>
</tr>
<tr>
<td>ROR</td>
<td>53%</td>
<td>52%</td>
<td>82%</td>
</tr>
<tr>
<td>AODU</td>
<td>-</td>
<td>-</td>
<td>75%</td>
</tr>
<tr>
<td>AODS</td>
<td>-</td>
<td>-</td>
<td>0%</td>
</tr>
<tr>
<td>AORS</td>
<td>-</td>
<td>-</td>
<td>54%</td>
</tr>
<tr>
<td>COR</td>
<td>-</td>
<td>-</td>
<td>100%</td>
</tr>
<tr>
<td>COD</td>
<td>-</td>
<td>-</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 26 Percentage of detected and live mutants per mutation operator
variables and change their values and that's where the fault can be caught some times.

- JID (member variable initialization deletion): In this type of mutants the initialization of an attribute is deleted from the constructor. An example is the private attribute (type double) throttle (private double throttle = 0.0). In this particular case, the value was initialized (to the same value of 0) later in test cases, specifically in the engineOff() function so the mutant could not be detected. But in cases where the sequence of functions calls did not provide an alternative initialization to the variable the fault can be detected.

- JSD (static modifier deletion): In this type of mutants the “static” modifier that signifies that a member belongs to the class itself rather than to instances of the class is deleted. These mutants were all caught by the compiler because the compiler “Cannot make a static reference to the non-static field”.

- EAM (Accessor Modifier Change): In this type of mutants an accessor method name is changed for other accessor method names provided that the signatures are the same (i.e, cs.getSpeed() is changed into cs.getBrakepedal()). These mutants are difficult to detect. Most of them made changes to the speed controller class attribute values that can be observed only if enough time is given to the system to execute. This class has a thread that starts when cruising is enabled to monitor the speed and adjust the throttle accordingly. The thread runs until the cruise control is disabled. In order to capture this type of faults we need a long execution time in which the controller is disabled and enabled many times so the speed calculations can take place while car is cruising (thread is running). These long executing test cases can only be found in the full coverage test suite. These sequences exist between states “cruising” and “standby”, where there are ten round trips (for those two states) that could be reached using different prefixes making 18 different possible sequences. These sequences are of length 5 – note that 5 is the maximum path length for the CruiseControl - Consider for example the following sequence: [(initial node) -> create -> (idle) -> engineOn -> (running) -> on -> (cruising) -> break -> (standby) -> on-> (cruising)]. The controller state changes three times: it is enabled after (on), disabled after (break), and re-enabled after the second (on)
event. Each time the controller state is enabled the current speed is recalculated based on the latest recorded speed. Every time these calculations take place there is a high chance of detecting the faults that caused wrong calculation to the recorded speed and/or relatively to the current speed. (For example the recordSpeed() function should set the recorded speed by calling cs.getSpeed() function that returns the current speed of the car. But instead the mutant change the cs.getSpeed() into getBrakepedal() which returns the current breakpedal level that can be of value=0. This fault can be caught because the car is running (in cruising state) and we are expecting the speed to be greater than 0).

- AODS (delete short-cut arithmetic operation): In this type of mutants the short-cut arithmetic operators are deleted. The mutation operator did not change the behaviour (returned value) of the method where the operator was applied.

4.5 Analysis of Code Coverage

CodeCover$^4$ is an extensible open source code coverage tool under Eclipse public licenses. We used CodeCover to measure the code coverage for each test driver / test case for the BFS/DFS and full coverage trees. (Note that we also used CodeCover to evaluate our new algorithm: see Chapter 5.)

We used all the available coverage criteria in CodeCover, specifically statement (S), branch (B), loop (L), and condition (C) coverage on all classes (i.e., four classes for CruiseControl. and two classes for OrderedSet—section 3.1.1 and 3.1.2). The four coverage criteria were not applicable for all classes (e.g., for the Overflow class only statement coverage is applicable since its implementation does not have any branching mechanism).

$^4$ http://www.codecover.org/documentation/tutorials/how_to_complete.html
Table 27 and Table 28 show the percentage of coverage of each test driver for both OrderedSet and CruiseControl respectively. The tables also show the total number of covered items (statements, branches, loops, and conditions) for each test driver.

Notice that the tables show the results obtained with the new algorithm. These will be discussed later.

<table>
<thead>
<tr>
<th>Test driver &amp; score</th>
<th>OrderedSet class</th>
<th>Overflow class</th>
<th>Number of covered items</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Statement</td>
<td>Branch</td>
<td>Loop</td>
</tr>
<tr>
<td>DFS tree 1 (76%)</td>
<td>73%</td>
<td>61.4%</td>
<td>58.3%</td>
</tr>
<tr>
<td>DFS tree 2 (74%)</td>
<td>73%</td>
<td>61.4%</td>
<td>58.3%</td>
</tr>
<tr>
<td>DFS tree 3 (69%)</td>
<td>73%</td>
<td>61.4%</td>
<td>58.3%</td>
</tr>
<tr>
<td>DFS tree 4 (75%)</td>
<td>73%</td>
<td>61.4%</td>
<td>58.3%</td>
</tr>
<tr>
<td>DFS tree 5 (73%)</td>
<td>75.7%</td>
<td>65.9%</td>
<td>58.3%</td>
</tr>
<tr>
<td>DFS tree 6 (73%)</td>
<td>73%</td>
<td>61.4%</td>
<td>58.3%</td>
</tr>
<tr>
<td>DFS tree 7 (73%)</td>
<td>73%</td>
<td>61.4%</td>
<td>58.3%</td>
</tr>
<tr>
<td>BFS tree 1 (78%)</td>
<td>75.7%</td>
<td>65.9%</td>
<td>58.3%</td>
</tr>
<tr>
<td>BFS tree 2 (78%)</td>
<td>75.7%</td>
<td>65.9%</td>
<td>58.3%</td>
</tr>
<tr>
<td>Full Coverage (66%)</td>
<td>68.9%</td>
<td>61.4%</td>
<td>54.2%</td>
</tr>
<tr>
<td>NewAlgorithm tree 1 (80.5%)</td>
<td>79.7%</td>
<td>68.2%</td>
<td>62.5%</td>
</tr>
</tbody>
</table>

Table 27 Code coverage percentages of test drivers (OrderedSet)

<table>
<thead>
<tr>
<th>Test driver &amp; score</th>
<th>CarSimulator class</th>
<th>Controller class</th>
<th>CruiseControl class</th>
<th>SpeedControl class</th>
<th>Number of covered items</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S</td>
<td>B</td>
<td>L</td>
<td>C</td>
<td>S</td>
</tr>
<tr>
<td>DFS/BFS tree 1 (50%)</td>
<td>83</td>
<td>71</td>
<td>66</td>
<td>50</td>
<td>95</td>
</tr>
<tr>
<td>DFS/BFS tree 2 (45%)</td>
<td>81</td>
<td>67</td>
<td>66</td>
<td>57</td>
<td>95</td>
</tr>
<tr>
<td>DFS/BFS tree 3 (42%)</td>
<td>77</td>
<td>60</td>
<td>66</td>
<td>28</td>
<td>95</td>
</tr>
<tr>
<td>Full Coverage (53%)</td>
<td>83</td>
<td>71</td>
<td>66</td>
<td>50</td>
<td>95</td>
</tr>
<tr>
<td>NewAlgorithm tree (52%)</td>
<td>85</td>
<td>76</td>
<td>66</td>
<td>57</td>
<td>95</td>
</tr>
</tbody>
</table>

Table 28 Code coverage percentages of test drivers (CruiseControl)

From Table 27 (OrderedSet) we can notice that BFS trees have higher code coverage than DFS trees, and the full coverage tree has lower code coverage than both DFS/BFS trees. From Table 28 (CruiseControl) we can see that the full coverage tree has the same coverage as DFS/BFS tree 1, which appears to be the best tree in terms of coverage. We
also notice that the largest variation of coverage percentages appear in the largest class, specifically class CarSimulator (148 LOC).

We used the t-test for two unpaired samples to check the significance of the relation between mutation scores of transition trees and the total number of covered items. By using the information provided in Table 27 and Table 28 we can calculate $t=47.22$, and $t=53.05$. Using t-distribution tables the two-tailed $p$ value is less than 0.0001. This difference is considered to be statistically significant and the higher code coverage percentage achieved the higher mutation score. Figure 18 show a scattered diagram that illustrates the positive effect of coverage percentage on the mutation score.

![Figure 18 Correlation between code coverage percentages and mutation score](image)

The positive correlation between mutation score and the percentage of code covered explains the high score recorded by the random test suites (section 4.3.1) that were mainly intended to achieve high coverage, although they were restricted by limited the number of paths and length of paths.

### 4.6 Summary

From the previous studies we can conclude that: (1) The number of paths, the length of paths and the number of transitions have positive effects on mutation score of a test suite; (2) The variability of states and events could be one of the reasons for the high
scores achieved by the path; (3) Covering round trip paths do not affect mutation score significantly. Based on these findings we proposed a new algorithm that aim to cover all the transitions of the state machine while generating all possible long paths that ensure a high variability of both events and states. The algorithm does not aim to necessarily cover round trip paths.
Chapter 5 New Algorithm

Based on the previous results that clarify the significant impact of some factors on the mutation score such as the number of paths, the number of transitions, the length of paths, the type of paths, and the sequence of events, we proposed a new algorithm aiming to reduce the cost of the generated test suite without affecting the fault detection effectiveness. The main principle of the new algorithm is therefore to get all the possible longest paths for a given graph, using depth first algorithm, such that every pair (triggering event, target state) appears only once in each path, in an attempt to obtain paths with the highest possible diversity of states and events. This eliminates the negative impact of repeated events and states and limits the generation of paths of length one. Section 5.1 presents our algorithm and we study its cost and effectiveness in section 5.2.

5.1 The new algorithm

Figure 19 shows, step by step, our new algorithm for generating transition trees. Our main goal when applying the new algorithm was to prevent the repetition of certain combination of both events and state to achieve high variability in path and therefore tree.

In this algorithm we are trying to build paths in a stepwise manner by hitting a new state that was not visited before in the path (step 2.a). If no new state is available, we hit any state with a (new) transition that was not used before in the path: step 2.b. (This ensures we increase variability.) Note that step 2.b allows the selection of the state of the
current node as the state of the new node, as long as the transition is new. If neither is possible, then we consider the path complete (the current node of the path, which is a leaf node of the path, is a terminal node) and we backtrack to build a different path, unless all the paths are complete, in which case the transition tree is completed and we stop.

**New Algorithm for generating transition trees**

1. Create a root node in the tree that is the initial state of the state machine. The current node is the root node.
2. Examine the outgoing transitions from the state corresponding to the current node that are not already visited in the whole tree. (If they have all been visited then go to step 5.) Do either one of the following, in the order they are presented below:
   a. Select an outgoing transition which target state is not already visited in the path from the root node to the current node. Draw a branch from the current node to a node representing that target state of that transition, labeled with the transition’s triggering event. Jump to step 3.
   b. [If no transition can be selected in point a.] Select an outgoing transition (triggering event, target state) such that the pair event/target state is not already somewhere (else) in the tree. Draw a branch from the current node to a node representing that target state of that transition, labeled with the transition’s triggering event. Jump to step 3.
3. Set the target node of the newly created branch as the current node and repeat from step 2.
4. (You have reached a leaf node in the tree. i.e., the end of a path in the tree.) If the current node is the root node, jump to step 6. Else, go to step 5.
5. Back track by setting the source node of the lastly created branch (in the current path) as your current node. Repeat from step 2.
6. Stop.

**Figure 19 New Algorithm for generating transition trees**

This new algorithm can produce multiple trees, depending on the traversal of the graph, similarly to BFS/DFS algorithms. Appendix G shows one of the possible generated trees using the new algorithm for the each case study.

**5.2 Evaluating the new algorithm**

The high diversity of states and events cannot always be achieved by the algorithm because of the nature of the state machine. As stated earlier in Table 1 the CruiseControl
has four states and 28 events, while the OrderedSet has eight states and 32 events. The
nature of the state machine of OrderedSet (i.e., large number of events) limited our
capacity to obtain a high variation in terms of events per path (e.g., we observed an event
can be reused up to 5 times per path) and to avoid paths of length one. Similarly, the
nature of the state machine of CruiseControl (i.e., small number of events) limited our
capacity to obtain high variation in terms of states in paths (e.g., we observed that a state
can be reused up to 5 times per path).

For both case studies we achieved a high variation of the combination of states and
events in paths and in the tree. (E.g., the same type of state was never hit by the same
type of event except in few cases).

In order to evaluate the cost-effectiveness of the trees generated using the new
algorithm we compared their structure and mutation scores with the best scored DFS and
BFS trees. This comparison is shown in Table 29.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>DFS</th>
<th>BFS</th>
<th>Full coverage</th>
<th>New Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trees</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nodes</td>
<td>34</td>
<td>34</td>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>Edges (no duplicates)</td>
<td>33</td>
<td>33</td>
<td>34</td>
<td>33</td>
</tr>
<tr>
<td>Edges (with duplicates)</td>
<td>97</td>
<td>57</td>
<td>52</td>
<td>55</td>
</tr>
<tr>
<td>Paths</td>
<td>25</td>
<td>25</td>
<td>21</td>
<td>14</td>
</tr>
<tr>
<td>Depth</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>Mutation score</td>
<td>76%</td>
<td>78%</td>
<td>66%</td>
<td>80.5%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>BFS/DFS</th>
<th>Full coverage</th>
<th>New Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trees</td>
<td>T1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nodes</td>
<td>30</td>
<td>35</td>
<td>29</td>
</tr>
<tr>
<td>Edges (no duplicates)</td>
<td>29</td>
<td>35</td>
<td>28</td>
</tr>
<tr>
<td>Edges (with duplicates)</td>
<td>89</td>
<td>111</td>
<td>41</td>
</tr>
<tr>
<td>Paths</td>
<td>25</td>
<td>28</td>
<td>9</td>
</tr>
<tr>
<td>Depth</td>
<td>5</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>Mutation score</td>
<td>50%</td>
<td>53%</td>
<td>52%</td>
</tr>
</tbody>
</table>

Table 29 Comparison of Characterizes and mutation scores between DFS/BFS trees
and the new algorithm trees
For the OrderedSet, the new algorithm achieved higher score (5.9% higher) than the best DFS tree, higher score (3.2% higher) than the best BFS tree, and higher score (21.9% higher) than Full coverage. In addition, the cost in terms of number of paths was reduced by 44% compared to DFS/BFS trees, and 33% compared to full coverage. The number of edges (with duplicates) was reduced by 43% compared to DFS trees, 3.5% compared to BFS trees, but 5.7% increase compared to Full coverage. For the CruiseControl, the new algorithm achieved higher score (4% higher) than the best DFS/BFS tree but lower score (1.8% lower) than full coverage. The cost in terms of number of paths was reduced by 64% compared to BFS/DFS trees, and 67% compared to Full coverage. In terms of number of edges (with duplicates) the reduction was 53.9% compared to DFS/BFS trees and 63% compared to Full coverage.

The trees produced by the new algorithm have the expected good properties: long paths, low repetitions of events and/or states in paths. Although none of the paths of the new algorithm trees were able to reach high scores like the paths of DFS/BFS trees, the total mutation score of the trees were high because each path catches unique faults. This can be observed in Table 30: for CruiseControl, DFS/BFS tree paths of length 5 reached an average score of 85% while path of the new algorithm scored 68% on average. The same is observed for OrderedSet DFS trees paths of length 7, which scored 70% on average while new algorithm paths scored 66% on average. Since individual paths of identical length score less and the overall score is higher, this means the individual paths overlap less in terms of the faults they reveal.
In addition, the code coverage of the new algorithm trees is higher than DFS/BFS trees and full coverage suites as shown in Table 27 and Table 28. For the OrderedSet and CruiseControl the new algorithm trees covered the highest number of items (158, and 253 respectively, i.e., 4% and 2.8% more than the average coverage of other algorithms, respectively) in comparison to DFS/BFS trees and full coverage. This result was expected because the new algorithm mainly aim to reach the highest possible variations of states and events, and therefore potentially triggers more combinations of behaviours which should exercise the code better.

Since the new algorithm has the highest code coverage among all test drivers, we wanted to see more precisely how the trees produced by the new algorithm compare to the other trees in terms of coverage. We were interested in assessing the percentage of shared coverage between the new algorithm trees and the other trees. CodeCover indicates that the new algorithm trees covers up to 100% of what the DFS trees cover, 96.4% of what BFS trees cover, and 96% of what the Full Coverage suite covers for OrderedSet, and 100% of what DFS/BFS trees cover, and 98% of what full coverage
covers for CruiseControl: Table 31 (covered items include statements, branches, loops, and conditions).

<table>
<thead>
<tr>
<th>New Algorithm</th>
<th>Shared percentage of covered items</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DFS</td>
</tr>
<tr>
<td>Cruise control</td>
<td>100%</td>
</tr>
<tr>
<td>OrderedSet</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 31 shared percentage of covered items between new algorithm trees, DFS/BFS trees and full coverage

The new algorithm trees were able to detect mutants from the same mutation operators as DFS/BFS trees, but were not able to detect mutants from any new mutation operator that could not be detected by DFS/BFS trees. Table 32 shows the score for each mutation operator. By comparing these scores to what we have in Table 26 we notice that for the CruiseControl some scores dropped by 13% (on average) for operators like JSD, and LOI and other scores increased up to 100% for operators like AORB. Some of the arithmetic mutants are related to time-dependant variables (i.e. car speed) as mentioned in section 4.2. The high ability of the new algorithm tree in detecting these types of arithmetic mutants can be explained by the long test sequences of the new algorithm tree that provide longer time of execution that raise the probability of detecting these types of faults. Some of these mutants made changes to values of class attributes that depend on time and can be observed only if enough time is given to the system to execute For example, the speed controller thread is continuously checking the current speed of the car every 500 milliseconds, and it adjusts both the throttle and the relative speed controller value based on the current speed. Some of the mutants like AOIs that Insert short-cut arithmetic operator adds minus operator to the calculated speed. With a short execution time the speed may not have a chance to change or increment (while the mutant is
supposed to decrement the speed). And thus the wrong adjustment of the throttle and the speed controller values may not be detected.

For the OrderedSet some scores increased by 22% on average for operators like AORB, LOI, AODU, and AORS. Recall that the new algorithm tree covers all transitions and the full coverage suite is missing two transitions. The lower ability of the full coverage suite in detecting some arithmetic mutants is due to missing two sequences as mentioned in section 4.1. These two sequences where the only sequences that cause the set to resize, and thus the full coverage tree misses many mutants that exists in the resizeArray() function.

| Class | Mutation Operator | CruiseControl | | | | OrderedSet | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Trees | Full Coverage | New algorithm tree | Trees | Full Coverage | New algorithm tree | | | | | |
| | JID | 22% | 0% | 22% | - | - | - | | | | |
| | JSD | 100% | 0% | 75% | Compile | Compile | Compile | | | | |
| | JSI | 0% | 9% | 0% | 0% | 0% | 0% | | | | |
| | EAM | 0% | 21% | 0% | 0% | 0% | 0% | | | | |
| | JDC | 0% | 0% | 0% | - | - | - | | | | |
| Traditional | AOIS | 52% | 60% | 50% | 75% | 74% | 75.5% | | | | |
| | AORU | 50% | 48% | 40% | 81% | 65% | 82% | | | | |
| | AORB | 13% | 13% | 100% | 80% | 73% | 100% | | | | |
| | ASRS | 67% | 67% | 66% | - | - | - | | | | |
| | COI | 100% | 100% | 100% | 100% | 75% | 100% | | | | |
| | LOI | 64% | 67% | 60% | 87% | 72% | 90% | | | | |
| | ROR | 53% | 52% | 53% | 82% | 82% | 82% | | | | |
| | AODU | - | - | - | 75% | 75% | 100% | | | | |
| | AODS | - | - | - | 0% | 0% | 0% | | | | |
| | AORS | - | - | - | 54% | 45% | 63% | | | | |
| | COR | - | - | - | 100% | 50% | 100% | | | | |
| | COD | - | - | - | 100% | 100% | 100% | | | | |

**Table 32 Percentage of detected mutants for the new algorithm trees**

From Table 32 we can notice that the new algorithm scored less than other algorithms for some mutants such as AOIS, AOIU, and LOI (for CruiseControl). These lower scores are due to some sub/sequences, required to kill mutants, that were absent from the tree generated by the new algorithm.
For example, some of the mutants created for AOIU operator (Arithmetic Operator Insertion-binary) change speed or distance variable values (fdist is changed into -fdist, fspeed is changed into -fspeed) by adding minus operator. These types of mutant can be captured by a sequence that requires enabling the cruise controller after it has been disabled and recalculating the speed such as [(cruising) -> break-> (standby) -> on -> (cruising)]. In this scenario the controller will be enabled and the current car speed is recorded to be maintained during cruising, and since the latest speed and/or distance were not correctly recorded because of the negative operator added by the mutant, the fault is raised and can be caught – i.e., distance decrement while car running.

Another typical example is one created by the LOI (Logical operator insertion) mutants operator, that adds a unary operator to the variable (e.g., -brakepedal). This mutant can be killed only by one sequence in tree2 [break-> (standby) -> accelerate -> (standby)]. In this sequence when break occurs, the value of brakepedal is set to 1, and when accelerate occurs, the value of brakepedal is checked and should be set to 0. But because the mutant adds a bitwise complement to the brakepedal in the checking statement [if (~brakepedal > 0)] the checked value is 0 but the original value of brakepedal remains 1. In this example the car is accelerating and breaking at the same time. These scenarios were missed by the new algorithm because the algorithm is mainly trying to target a new state whenever possible, and less frequently targets the same state, while these scenarios require the algorithm to back track to previously visited state, which may very seldom occur when applying the new algorithm - as we were avoiding the repetition of states in the path-.
Still these scenarios that requires targeting the same state or back tracking to previous state could be covered in others possible trees generated by the new algorithm.

Figure 20 and Figure 21 show the results presented in Percentage of detected mutants for the new algorithm treesTable 32Table 29. Figure 20: distributions of the scores of DFS/BFS and the two trees generated by the new algorithm. It shows the high mutation scores achieved by the new trees compared to the DFS/BFS trees. Figure 21 shows the distribution of the number of paths of DFS/BFS trees and the two new trees. It shows the small number of paths of the two new trees compared to the DFS/BFS trees.

Figure 20 Unified scale distribution histograms of mutation scores of all trees

Figure 21 Distribution histograms of number of paths generated from all trees
Chapter 6 Threats to Validity

As usual in any experiment, possible threats to validity have to be considered. Threats to conclusion validity are concerned with issues that affect the ability to draw the correct conclusion. In our case, they could be essentially due to low statistical power, resulting sometimes from the small number of transition trees. For this reason we did not rely too much on statistical results and complemented the quantitative analysis with a qualitative analysis. An internal validity threat exists when the outcome of the experiment may not necessarily be caused by the treatment applied, but can be caused by another factor not controlled in the experiment. We suggest possible causes for the phenomenon we observed. Obviously, other reasons could explain our observations. More case studies would be necessary to limit further this threat. Construct validity is mainly related to our use of mutants to measure the fault detection effectiveness. However, the existing literature suggests that faults seeded using mutation operators can be representative of real faults, and relying on mutation to compare test techniques is practical as it provides large, automatically generated fault samples. The fact that we used two class clusters with very different code characteristics—thus leading to very different samples of mutants—should limit the likelihood of this threat. External validity relates to the external aspects that interact with the treatments and limit the generalization of the results. This threat is mainly related to our selection of case study systems. Although we carefully selected representative systems, more case studies are necessary to further limit this threat.
Chapter 7 Conclusions

This research investigated, through an experiment, the cost and fault-detection effectiveness of the round trip path strategy for state based testing. We compared the full round trip coverage strategy to the partial round trip coverage presented in transition trees. The partial coverage represented in transition trees covers 76.6% on average of the total number of round tips in a UML state machine. The results show that: (1) the full coverage of round trips does not have better fault detection effectiveness than the partial coverage of round trips, which indicates that coverage level of the round trip criterion is not an adequate measure for selecting the best test suite; (2) Although test suites sizes were much converged in terms of number of paths, there is sensible variation of mutation scores, which implies that test suite size have a significant positive effect on mutation score; (3) Covering new transition means more code coverage but covering the same transitions several times does not (necessarily) mean more fault detection but rather a useless addition to the cost. Thus a large number of transitions in a tree is not a sign for better mutation score; (4) the algorithm does not significantly affect the mutation score of the transition trees it can generate.

Furthermore we investigated the factors that can influence the effectiveness of a test case (i.e., transition tree path), and hence the mutation scores of the test suite such as: path length, path type, and state-events sequences.

We observed that the effectiveness of the round trip criterion is affected by:
- The nature of the system under test,
- The characteristics of the UML state machine (i.e., number of events and transitions),
- The methodology/algorithm used to derive the round trips, that affects the size of the test suite, the type of paths, and sequence of events.
- The input values—if any—used in the test drivers, that affect the code coverage.

We suggested some feasible improvements that can be applied to the round trip coverage aiming to avoid the observed limitations with the aim of achieving higher mutation score such as: (1) increasing the variation events and states visited in a path to help achieve better code coverage (and therefore achieve better fault detection), (2) the choice of longest (as long as possible) paths to increase variations.

Based on the experiment results, we proposed a new algorithm that can improve round trip path techniques. We aimed to address the high cost of the generated test suites by eliminating repeated sequences of state-events and maintaining long path, without affecting fault detection effectiveness. The application of the new algorithm on the two case studies showed a high cost reduction that reach up to 67% in terms of path number and up to 63% in terms of the number of transitions. The sequences of the generated test cases were able to reveal unique faults and 3.5% improvement in mutation scores (on average) was recorded.

Our future work will address the discussed threats by performing similar experiments on other systems. Other work can investigate: optimum levels of round trip coverage; optimal ratio between prefix and round trip portion; impact of state-machine graph characteristics, and criteria that can help determining the best transition tree.
References


[17] Sunwoo Kim, John A.Clark, John A.McDermid “Class Mutation: Mutation Testing for Object-Oriented Programs” University of York, United Kingdom


[20] Yu-Seung Ma, Yong-Rae Kwon, Jeff Offutt “An Experiment Mutation Operator for Java”ACM ,Volume 29 No.5, September 2007


[22] Sunwoo Kim, John A.Clark, John A.McDermid “Class Mutation: Mutation Testing for Object-Oriented Programs” University of York, United Kingdom


Appendix A Depth-First and Breadth-First Algorithms

A common representation form was used to represent the UML state machine graph as one dimensional matrix; where the row index represents the starting node for a transition, the columns index represents the end node for a transition, and the cell number represents the transition number itself. The algorithm generates a tree by extracting all the outgoing transitions for a certain node from the matrix, and then checks each of the next target nodes for these transitions. If the target node was not visited before it will be branched. But if the next target node was already visited before no more branching for this node goes on and the algorithm backtracks to the parent node. The process continues until all the nodes in the graph are visited (they are terminal).

Because there might be more than one possibility to branch from a node in the same level of the tree, a new tree will be created for each possibility of branching from a node in a new thread depending on the next successor node to be branched.

The termination criteria adopted is the one proposed by Binder; the node is considered terminal (can not be branched any more) if it has no outgoing edge (final state in the state machine), or if the state it represents has already been drawn (branched before), or if it is the final (ω) state.

The algorithm may generate redundant trees (redundant as defined in section 3.2.1). In order to omit redundant trees, a list of checks was made on the node before generating the new tree:

1. If the target node of this transition already has no outgoing edges
2. If the target node of this transition is terminal "not visited before" do not create new thread
3. If the target node of this transition is the "only" not visited child node of the parent node of this transition
4. if the target node of this transition is the "first" not visited child node of the parent node of this transition
5. Only for the Main (first) thread if the start node was the initial node, propagation stops.

If any of these checks was false no new thread created to generate new tree. All of these checks should be true in order to create a new tree in new thread.

The algorithm generating all possible depth-first traversals is in Figure 22, the algorithm generating all possible breadth-first traversals is in Figure 23, while Figure 24 shows a class diagram design of the traversal algorithms.

Although a great care was given when developing the algorithms to avoid the generation of redundant trees (trees with the same set of transition paths), we checked, once all trees were generated, and that the set of trees did not contain duplicates.

---

**Algorithm for generating transition trees using depth first**

**Procedure 1: Void node** (CGN: graphNode, CTN: treeNode)

Do{
  n number of outgoing not visited transitions of CGN
  if (n=0)
  
  
  CTN=pop;
  CGN=pop;
  Continue;
  
  if (not one of the outgoing transitions of CGN is visited)
  
  for each of the 2 to n outgoing transition i of CGN
  
  Check(1) if the target node of the outgoing transition was not visited
  
  Check(2) if the target node of the outgoing transition has no outgoing edges
  
  Check(3) If the target node of the outgoing transition (is the "only" not visited child node for the parent node of this transition)
  
  Check(4) If the target node of the outgoing transition is the "first" not visited child node for the parent node of this transition
  
  Check(5) If the Parent node of the outgoing transition was the initial node and the initial node has more than one outgoing edge and the current thread is the first thread
  
  Create new thread (copy of tree; copy of stack, copy of visited nodes, and visited transitions)
  
  Tell new thread to continue traversal from transition I

---

109
Call visit transition (CGN, transition I, CTN)
}
t=get next transition to visit (CGN)
If (t=null)
{CTNNode = pop ; CGNNode = pop ; continue; }
newTNode = target node of t
Add t to visited transitions
newTEdge from CTN to newTNode
If (newTNode already visited)
{continue;}
Add newTNode to visited nodes
Push CGN, CTN
CGN = target of t
CTN = newTNode
}While (stack not empty)
Print StringBuilder to screen
Procedure 2: Visit Transition (CGN: grapgNode, t:transition, CTN:treeNode)
{newTNode = target of t
NewTEdge = from CTN to newTNode
Add t to visited transitions
If (newTNode already visited)
{
 CTN = pop
 CGN = pop
 VisitNode (CGN,CTN)
}
Else
Push CGN;
Push CTN;
VisitNode(target of t , newTNode) }
Procedure : main
CTN = create a tree node for the root of the graph
CGN= root of the graph
Add CGN to visited nodes
Call VisitNode (CGN, CTN)

Figure 22 Algorithm for generating transition trees using depth first

Algorithm for generating transition trees using Breadth first

Procedure 1: Void node (CGN: graphNode, CTN: treeNode)
Do{
 n number of outgoing not visited transitions of CGN
 if (n=0)
{  
    CTN=pop ;
    CGN=pop;
  Continue;
 }
If ( not one of the outgoing transitions of CGN is visited )
{}
for each of the 2 to n outgoing transition I of CGN
{
  Check(1) if the target node of the outgoing transition was not visited
  Check(2) if the target node of the outgoing transition has no outgoing edges
  Check(3) If the target node of the outgoing transition

110
(is the "only" not visited child node for the parent node of this transition)
Check(4) If the target node of the outgoing transition is the "first" not visited child node for the parent node of this transition
Check(5) If the Parent node of the outgoing transition was the initial node and the initial node has more than one outgoing edge and the current thread is the first thread
Create new thread (copy of tree; copy of stack, copy of visited nodes, and visited transitions)
Tell new thread to continue traversal from transition I
Call visit transition (CGN, transition I, CTN)

 Procedure 2: Visit Transition (CGN: grapgNode, t:transition, CTN:treeNode)
{ newTNode = target of t NewTEdge = from CTN to newTNode Add t to visited transitions If (newTNode already visited) {continue;} Add newTNode to visited nodes Push CGN, CTN CGN = target of t CTN = newTNode }
While (stack not empty) Print StringBuilder to screen
Procedure : main
CTN = create a tree node for the root of the graph
CGN= root of the graph
Add CGN to visited nodes
Call VisitNode (CGN, CTN)

Figure 23 Algorithm for generating transition trees using breadth first
Figure 24 Class Diagram (UML state machine traversal)
Appendix B CruiseControl Class Diagram

Figure 25 Class Diagram (CruiseControl)
Appendix C Transition trees for OrderedSet

<table>
<thead>
<tr>
<th>Event</th>
<th>Number</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>OrdSet(int[2]v)</td>
<td>1</td>
<td>Initial Node</td>
</tr>
<tr>
<td>OrdSet(int[1]v)</td>
<td>2</td>
<td>Empty</td>
</tr>
<tr>
<td>OrdSet(x)</td>
<td>3</td>
<td>Filled (F2,F4,F6)</td>
</tr>
<tr>
<td>OrdSet(int[1]v)</td>
<td>4</td>
<td>Partially Filled (PF1,PF3,PF5)</td>
</tr>
<tr>
<td>OrdSet(int[3]v)</td>
<td>5</td>
<td>Overflow</td>
</tr>
<tr>
<td>OrdSet(int[5]v)</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>OrdSet(int[7]v)</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>remove(x)</td>
<td>8, 10, 18, 25, 32</td>
<td></td>
</tr>
<tr>
<td>remove(val)</td>
<td>16, 23, 30</td>
<td></td>
</tr>
<tr>
<td>remove(n)</td>
<td>11, 15, 21, 28</td>
<td></td>
</tr>
<tr>
<td>add(x)</td>
<td>9, 33</td>
<td></td>
</tr>
<tr>
<td>add(val)</td>
<td>12, 14, 19, 22, 26, 29</td>
<td></td>
</tr>
<tr>
<td>add(n)</td>
<td>13, 17, 20, 24, 27, 31</td>
<td></td>
</tr>
</tbody>
</table>

Table 33 Legend for transition trees
Figure 26 Transition tree 1 (OrderedSet). Figure 27 Transition tree 2 (OrderedSet).
Figure 28 Transition tree 3 (OrderedSet).

Figure 29 Transition tree 4 (OrderedSet).
Figure 30 Transition tree 5 (OrderedSet).

Figure 31 Transition tree 6 (OrderedSet).
Figure 32 Transition tree 7 (OrderedSet).
Appendix D Transition Trees for CruiseControl

Figure 33 Transition tree1 (CruiseControl)
Figure 34 Transition tree2 (CruiseControl)
Figure 35 Transition tree3 (CruiseControl)
Appendix E Transition Trees Covering Round Trip Paths Fully

Sections E.1 and E.2 give the transition tree paths for the transition trees covering round trip paths fully for our two case studies: round trip paths are identified, along with the corresponding triggering events, as well as their prefix (in bold face).

**E.1 Sequence of all round trip paths for CruiseControl**

1. (Initial Node) -> create -> (Inactive/Idle) -> accelerator -> (Inactive/Idle)
2. (Initial Node) -> create -> (Inactive/Idle) -> brake -> (Inactive/Idle)
3. (Initial Node) -> create -> (Inactive/Idle) -> off -> (Inactive/Idle)
4. (Initial Node) -> create -> (Inactive/Idle) -> resume -> (Inactive/Idle)
5. (Initial Node) -> create -> (Inactive/Idle) -> engineOff -> (Inactive/Idle)
6. (Initial Node) -> create -> (Inactive/Idle) -> on -> (Inactive/Idle)
7. (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> engineOff -> (Inactive/Idle)
8. (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> engineOff -> (Inactive/Idle)
9. (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> off -> (Standby/Running) -> engineOff -> (Inactive/Idle)
10. (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> accelerator -> (Standby/Running) -> engineOff -> (Inactive/Idle)
11. (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> brake -> (Standby/Running) -> engineOff -> (Inactive/Idle)
12 - (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> resume -> (Active/Running)
13 - (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> brake -> (Active/Running)
14 - (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> accelerator -> (Active/Running)
15 - (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> off -> (Active/Running)
16 - (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> engineOn -> (Active/Running)
17 - (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> resume -> (Cruising/Running)
18 - (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> on -> (Cruising/Running)
19 - (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> engineOn -> (Cruising/Running)
20 - (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> off -> (Standby/Running) -> on -> (Cruising/Running)
21 - (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> off -> (Standby/Running) -> resume -> (Cruising/Running)
22 - (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> accelerator -> (Standby/Running) -> on -> (Cruising/Running)
23 - (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> accelerator -> (Standby/Running) -> resume -> (Cruising/Running)
24 - (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> brake -> (Standby/Running) -> on -> (Cruising/Running)
25 – (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> brake -> (Standby/Running) -> resume -> (Cruising/Running)

26 – (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> off -> (Standby/Running) -> engineOn -> (Standby/Running)

27 – (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> accelerator -> (Standby/Running) -> accelerator -> (Standby/Running)

28 – (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> brake -> (Standby/Running) -> off -> (Standby/Running)

29 – (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> off -> (Standby/Running) -> brake -> (Standby/Running)
E.2 Sequence of all round trip paths for OrderedSet

1 - (Initial Node) -> OrdSet(int[ ]v) -> (Empty) -> remove(x) -> (Empty)

2 - (Initial Node) -> OrdSet() -> (Empty) -> add(x) -> (Partially Filled 1) -> remove(val) -> (Empty) //node1

3 - (Initial Node) -> OrdSet(int[ ]v) -> (Partially Filled 1) -> remove(n) -> (Partially Filled 1)

4 - (Initial Node) -> OrdSet(int[ ]v) -> (Partially Filled 1) -> add(val) -> (Partially Filled 1)

5 - (Initial Node) -> OrdSet(int[ ]v) -> (Partially Filled 1) -> remove(val) -> (Empty) -> add(x) -> (Partially Filled 1)

6 - (Initial Node) -> OrdSet(int[ ]v) -> (Partially Filled 1) -> add(n) -> (Filled 2) -> remove(val) -> (Partially Filled 1)

7 - (Initial Node) -> OrdSet(int[ ]v) -> (Filled 2) -> remove(n) -> (Filled 2)

8 - (Initial Node) -> OrdSet(int[ ]v) -> (Filled 2) -> add(val) -> (Filled 2)

9 - (Initial Node) -> OrdSet(int[ ]v) -> (Filled 2) -> remove(val) -> (Partially Filled 1) -> add(n) -> (Filled 2) //node2

10 - (Initial Node) -> OrdSet(int[ ]v) -> (Partially Filled 3) -> remove(x) -> (Partially Filled 3)

11 - (Initial Node) -> OrdSet(int[ ]v) -> (Partially Filled 3) -> add(val) -> (Partially Filled 3)

12 - (Initial Node) -> OrdSet(int[ ]v) -> (Partially Filled 3) -> add(n) -> (Filled 4) -> remove(val) -> (Partially Filled 3) //node3

13 - (Initial Node) -> OrdSet(int[ ]v) -> (Partially Filled 3) -> add(n) -> (Filled 4) -> remove(n) -> (Filled 4)

14 - (Initial Node) -> OrdSet(int[ ]v) -> (Partially Filled 3) -> add(n) -> (Filled 4) -> add(val) -> (Filled 4) //node4

15 - (Initial Node) -> OrdSet(int[ ]v) -> (Partially Filled 5) -> add(val) -> (Partially Filled 5)

16 - (Initial Node) -> OrdSet(int[ ]v) -> (Partially Filled 5) -> remove(x) -> (Partially Filled 5)

125
17 – (Initial Node) -> OrdSet(int[5]v) -> (Partially Filled 5) -> add(n) -> (Filled 6) -> remove (val) -> (Partially Filled 5) //node6
18 – (Initial Node) -> OrdSet(int[5]v) -> (Partially Filled 5) -> add(n) -> (Filled 6) -> remove (n) -> (Filled 6)
19 – (Initial Node) -> OrdSet(int[5]v) -> (Partially Filled 5) -> add(n) -> (Filled 6) -> add (val) -> (Filled 6) //node7
20 – (Initial Node) -> OrdSet(int[7]v) -> (OverFlow) -> add(x) -> (OverFlow)
21 – (Initial Node) -> OrdSet(int[7]v) -> (OverFlow) -> remove(x) -> (OverFlow) //node8

E.3 Tree Structure of all round trip paths for OrderedSet

Figure 36 Tree Structure of all round trip paths (OrderedSet)
E.4 Tree Structure of all round trip paths for CruiseControl

Figure 37 Tree Structure of all round trip paths (CruiseControl)
Appendix F Random Test Suites

F.1 Sequence of random test suite for CruiseControl (6 L1, 6 L2, 4 L3, 13 L4)

1 – (Initial Node) -> create -> (Inactive/Idle) -> accelerator -> (Inactive/Idle)
2 – (Initial Node) -> create -> (Inactive/Idle) -> brake -> (Inactive/Idle)
3 – (Initial Node) -> create -> (Inactive/Idle) -> off -> (Inactive/Idle)
4 – (Initial Node) -> create -> (Inactive/Idle) -> resume -> (Inactive/Idle)
5 – (Initial Node) -> create -> (Inactive/Idle) -> engineOff -> (Inactive/Idle)
6 – (Initial Node) -> create -> (Inactive/Idle) -> on -> (Inactive/Idle)
7 – (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> resume -> (Active/Running)
8 – (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running)
9 – (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> engineOff -> (Inactive/Idle)
10 – (Initial Node) -> create -> (Inactive/Idle) -> off -> (Inactive/Idle) -> engineOn -> (Active/Running)
11 – (Initial Node) -> create -> (Inactive/Idle) -> brake -> (Inactive/Idle) -> resume -> (Inactive/Idle)
12 – (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> accelerator -> (Active/Running)
13 – (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> engineOff -> (Inactive/Idle)
14 – (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> resume -> (Cruising/Running)
15 – (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> off -> (Standby/Running)

16 – (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> break -> (Standby/Running)

17 – (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> off -> (Active/Running) -> on -> (Cruising/Running) -> engineOff -> (Inactive/Idle)

18 – (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> resume -> (Cruising/Running) -> off -> (Standby/Running)

19 –(Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> accelerometer -> (Standby/Running) -> resume -> (Cruising/Running)

20 – (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> engineOff -> (Inactive/Idle) -> break -> (Inactive/Idle)

21 – (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> off -> (Standby/Running) -> break -> (Standby/Running)

22 – (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> break -> (Standby/Running) -> engineOff -> (Inactive/Idle)

23 –(Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> accelerator -> (Standby/Running) -> off -> (Standby/Running)

24 – (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> off -> (Standby/Running) -> on -> (Cruising/Running)

25 – (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> break -> (Standby/Running) -> engineOn -> (Standby/Running)

26 – (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> engineOn -> (Cruising/Running) -> on -> (Cruising/Running)

27 – (Initial Node) -> create -> (Inactive/Idle) -> engineOn -> (Active/Running) -> on -> (Cruising/Running) -> engineOff -> (Inactive/Idle) -> engineOn -> (Active/Running)

28 – (Initial Node) -> create -> (Inactive/Idle) -> on -> (Inactive/Idle) -> resume -> (Inactive/Idle) -> accelerator -> (Inactive/Idle) -> off -> (Inactive/Idle)

29 – (Initial Node) -> create -> (Inactive/Idle) -> on -> (Inactive/Idle) -> engineOn -> (Active/Running) -> accelerator -> (Active/Running) -> engineOff -> (Inactive/Idle)
F.2 Sequence of random test suite for OrdenedSet (11 L2 path, 10 L3 path)

1 - (Initial Node) -> OrdSet(int[2]v) -> (Filled2) -> remove(val) -> (Partially Filled 1)
2 - (Initial Node) -> OrdSet(int[1]v) -> (Empty) -> remove(x) -> (Empty)
3 - (Initial Node) -> OrdSet(x) -> (Empty) -> add(x) -> (Partially Filled 1)
4 - (Initial Node) -> OrdSet(int[1]v) -> (Partially Filled 1) -> remove(val) -> (Empty)
5 - (Initial Node) -> OrdSet(int[1]v) -> (Partially Filled 1) -> add(n) -> (Partially Filled 2)
6 - (Initial Node) -> OrdSet(int[3]v) -> (Partially Filled 3) -> add(n) -> (Filled 4)
7 - (Initial Node) -> OrdSet(int[3]v) -> (Partially Filled 3) -> add(val) -> (Partially Filled 3)
8 - (Initial Node) -> OrdSet(int[5]v) -> (Partially Filled 5) -> add(n) -> (Partially Filled 6)
9 - (Initial Node) -> OrdSet(int[5]v) -> (Partially Filled 5) -> remove(x) -> (Partially Filled 5)
10 - (Initial Node) -> OrdSet(int[7]v) -> (Overflow) -> add(x) -> (Overflow)
11 - (Initial Node) -> OrdSet(int[7]v) -> (Overflow) -> remove(x) -> (Overflow)
12 - (Initial Node) -> OrdSet(int[2]v) -> (Partially Filled 2) -> remove(n) -> (Partially Filled 2) -> add(n) -> (Partially Filled 3)
13 - (Initial Node) -> OrdSet(int[2]v) -> (Filled 2) -> add(val) -> (Filled 2) -> remove(val) -> (Partially Filled 1)
14 - (Initial Node) -> OrdSet(int[1]v) -> (Empty) -> add(x) -> (Partially Filled 1) -> add(n) -> (Partially Filled 2)
15 - (Initial Node) -> OrdSet(x) -> (Empty) -> add(x) -> (Partially Filled 1) -> remove(val) -> (Empty)
16 - (Initial Node) -> OrdSet(int[1]v) -> (Partially Filled 1) -> add(val) -> (Partially Filled 1) -> add(n) -> (Partially Filled 2)
17 - (Initial Node) -> OrdSet(int[3]v) -> (Partially Filled 3) -> add(val) -> (Partially Filled 3) -> remove(x) -> (Partially Filled 3)
18 - (Initial Node) -> OrdSet(int[3]v) -> (Partially Filled 3) -> add(n) -> (Partially Filled 4) -> remove(val) -> (Partially Filled 4)
19 - (Initial Node) -> OrdSet(int[5]v) -> (Partially Filled 5) -> add(n) -> (Filled 6) -> remove(val) -> (Partially Filled 5)
20– (Initial Node) -> OrdSet(int[5]v) -> (Partially Filled 5) -> add (n) -> (Filled 6) -> add (n) -> (Overflow)

21– (Initial Node) -> OrdSet(int[7]v) -> (Overflow) -> add (n) -> (Overflow) -> remove (x) -> (Overflow)
Appendix G Transition trees of the New Algorithm

Figure 38 one of the possible generated transition trees using the new algorithm (OrderedSet)
Figure 39 one of the possible generated transition trees using the new algorithm (CruiseControl)

133
Appendix H Additional Hypotheses Analysis

H.1 Mutation score analysis for test suites for each case study

We apply the one tailed t-test on the data of the total mutation scores of DFS/BFS transition trees and the "full coverage" presented in Table 7 for each case study. The results of the analysis is shown in Table 34.

<table>
<thead>
<tr>
<th>Case Study</th>
<th>Sample Mean (µ)</th>
<th>StDev</th>
<th>Population Mean</th>
<th>n</th>
<th>DF</th>
<th>t-value</th>
<th>t_0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>CruiseControl</td>
<td>45.6</td>
<td>4.04</td>
<td>53</td>
<td>3</td>
<td>2</td>
<td>-3.14</td>
<td>2.92</td>
</tr>
<tr>
<td>OrderedSet</td>
<td>74.3</td>
<td>2.82</td>
<td>66</td>
<td>9</td>
<td>8</td>
<td>8.82</td>
<td>1.86</td>
</tr>
</tbody>
</table>

Table 34 Descriptive statistics of t-test for mutation scores for each case study

For the CruiseControl, since the calculated t-Value is less than the tabled value of t_0.5, we cannot reject the null hypothesis \( H_0: \mu \leq 45.6 \). From this result we can conclude that the sample mean of partial coverage is lower than the population mean.

For the OrderedSet, since the calculated t-Value = 8.82 is greater than the tabled value of t_0.5 = 1.86, we can reject the null hypothesis \( H_0: \mu \leq 74.3 \). From this result we can conclude that the sample mean of partial coverage is not lower than the population mean.

The difference between the two results may simply be due to the small number of BFS/DFS trees for CruiseControl.
## Appendix I Mutants Code Sample

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Mutant</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>JSI (static modifier insertion)</strong></td>
<td></td>
</tr>
</tbody>
</table>
| CruiseControl | `private int distance = 0;`  
| | `private static int distance = 0;` |
| OrdrerdSet | `private int _resized_times;`  
| | `private static int _resized_times;` |
| **JID (member variable initialization deletion)** | |
| CruiseControl | `java.lang.Thread engine = null;`  
| | `private double throttle = 0.0;` |
| OrdrerdSet | `private int controlState = INACTIVE;`  
| | `private int controlState;` |
| **JSD (static modifier deletion)** | |
| CruiseControl | `public static private int brakepedal = 0;`  
| | `private int brakepedal = 0;` |
| OrdrerdSet | `public static final int min_set_size = 2;`  
| | `public final int min_set_size = 2;` |
| **EAM (Accessor Modifier Change)** | |
| CruiseControl | `public synchronized int getState(){`  
| | `return sc.getState();}  
| | `return sc.getCruiseSpeed();}` |
| OrdrerdSet | `if (x.getSetLast() != _last) `  
| | `if (x.getSetSize() != _last)` |
| **AORs (Arithmetic operator insertion - Insert short-cut operator) | |
| CruiseControl | `if (brakepedal > 0) {`  
| | `brakepedal = 0;}`  
| | `if (++brakepedal > 0) {`  
| | `brakepedal = 0;}` |
| OrdrerdSet | `return binSearch(_set, _last + 1, n ) >= 0;`  
| | `return binSearch(_set, ++_last + 1, n ) >= 0;` |
| **AORb (Arithmetic operator replacement - Replace basic binary operator) | |
| CruiseControl | `if (throttle > 0.0) {`  
| | `throttle -= 0.5 / ticksPerSecond;`  
| | `if (throttle > 0.0) {`  
| | `throttle -= 0.5 + ticksPerSecond;` |
| OrdrerdSet | `int j = nElts - 1;`  
| | `int j = nElts * 1;` |
| **LOI (Logical operator insertion)** | |
| CruiseControl | `private int controlState = INACTIVE;`  
| | `private int controlState = ~INACTIVE;` |
| OrdrerdSet | `return binSearch(_set, _last + 1, n ) >= 0;`  
| | `return binSearch(_set, ~_last + 1, n ) >= 0;` |

Table 35 Sample Mutants Code