Assessing, Comparing, and Combining State machine-Based Testing and Structural Testing: A Series of Experiments

By

Samar Mouchawrab

A thesis submitted to
The Faculty of Graduate Studies and Research
In partial fulfillment of the requirements of the degree of
Doctor of Philosophy in Electrical Engineering

Ottawa-Carleton Institute of Electrical and Computer Engineering (OCIECE)
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6

May 2010

Your file Votre référence
ISBN: 978-0-494-70562-9
Our file Notre référence
ISBN: 978-0-494-70562-9

Abstract

A large number of research works have addressed the importance of models in software engineering, mainly in the design and testing of robust software systems, for example in automating the test of software systems. Although models have been proven to be helpful in a number of software engineering activities, such as providing a better medium for communication among designers and customers, there is still significant resistance to model-driven development in many software organizations. The main reason is that it is perceived to be expensive and not necessarily cost-effective. This study investigates one specific aspect of this larger problem. It addresses the impact of using state machines for testing class clusters that exhibit a state-dependent behavior on testing cost effectiveness when compared with structural testing, which is perhaps the most common systematic testing practice. More precisely, it reports on a series of controlled experiments that investigate the impact of state machine-based testing on fault-detection and cost-effectiveness. Based on previous work showing this is a good compromise in terms of cost and effectiveness, this study focuses on a specific state-based technique that is an adaptation of the W-Method for UML state machines: the round-trip paths coverage criterion. Code-based structural testing is compared to round-trip paths testing and their combination is investigated to determine whether they are complementary. A series of four controlled experiments are presented and discussed in this thesis. Differences between the results of the different experiments are highlighted and plausible explanations for these differences are provided. Results show that, even when a state machine models accurately the behavior of the cluster under test, no significant difference between the fault detection effectiveness of the two test strategies is observed. And in all cases, even when the state machine's representation of the

cluster's behavior is limited, the two test strategies are significantly more effective when combined. This implies that a cost-effective strategy could be to specify state machine-based test cases early on, execute them when the source code becomes available, and then complete them with structural test cases based on coverage analysis. A qualitative analysis is also presented, where the reasons for undetected faults are investigated and ways to improve the state machine-based testing of source code are derived. As an outcome of this analysis and based on the presented results, this thesis recommends a number of updates to the round-trip paths testing strategy in order to improve its fault-detection effectiveness and lower its cost.

Dedication

To the people that pushed, that pulled and that accompanied me on the way:

To my parents, for having always stayed behind me and encouraged me.

To Ali, my loving husband, to be there for me all the way.

To Yasmine and Hadi, my precious gifts, my reason for joy.

To my sister Souheir, my best friend.

To my dear bothers Khaled and Raed.

Thank you all for the unconditional love, guidance, and support that you have always given me, helping me to succeed and instilling in me the confidence that I am capable of doing anything I put my mind to. Thank you for everything. I love you!

Acknowledgments

It is a pleasure to thank the many people who made this thesis possible.

It is difficult to overstate my gratitude to my Ph.D. supervisor, Professor Lionel C. Briand. With his enthusiasm, his inspiration, and his great efforts to explain things clearly and simply, he helped to make a researcher out of me. Throughout my PhD studies, he provided encouragement, sound advice, good teaching, and lots of good ideas. Thanks for making this thesis possible by teaching me perseverance and to always reach high.

I would like to express my deep and sincere gratitude to my supervisor, Professor Yvan Labiche for the valuable advice, the constructive criticism and for providing numerous ideas. His wide knowledge and his logical way of thinking have been of great value for me and for this work.

I wish to express my warm and sincere thanks to Professor Massimiliano Di Penta for his detailed and constructive comments, and for his important support throughout this work. I would also like to thank him for giving me the opportunity to teach (for a week) in the department of Engineering, University Sannio. It was a wonderful and enriching experience.

I am indebted to my colleagues at the SQUALL laboratory for providing a stimulating and fun environment in which to learn and grow. I am especially grateful to Mike Sowka for giving wise advice, helping with various applications and being a great friend.

I wish to thank my entire family for providing a loving environment for me. My husband, my kids, my parents, my siblings, and my mother and father in-law who were particularly supportive.

Table of Contents

CHAPTER 1 INTRODUCTION **1**

CHAPTER 2 RELATED WORK **6**

2.1 STATE-BASED AND STRUCTURAL TESTING 6

2.2 EMPIRICAL STUDIES 9

CHAPTER 3 EXPERIMENT DESCRIPTION **14**

3.1 EXPERIMENT DEFINITION AND CONTEXT 14

3.2 EXPERIMENT PLANNING 19

3.2.1 Context selection 20

3.2.2 Research questions 21

3.2.3 Variable selection 23

3.2.4 Mutant seeding 25

3.2.5 Experimental procedure 28

3.2.6 Experiment design 31

3.3 EXPERIMENT OPERATION 32

3.3.1 Preparation and material 32

3.3.2 Execution 33

3.3.3 Data Collection 35

3.4 EXPERIMENT REPLICATION 35

3.5 OVERVIEW OF STATISTICAL ANALYSIS 39

CHAPTER 4 EXPERIMENTAL RESULTS **43**

4.1 IMPACT OF TEST TECHNIQUES ON FAULT DETECTION EFFECTIVENESS 44

4.1.1 Univariate analysis 44

4.1.2 Interaction effects 66

4.1.3 Summary 74

4.2 COMBINING TEST TECHNIQUES IMPACT ON FAULT DETECTION EFFECTIVENESS 75

4.2.1 Complementarity of testing techniques 77

4.2.2 Impact of combining test techniques on fault detection effectiveness 82

4.2.3 Impact of cluster characteristics on combining test techniques 86

4.2.4 Impact of combining test techniques on fault detection effectiveness across mutation operators 87

4.2.5 Summary 90

4.3 COST ANALYSIS OF TEST TECHNIQUES 91

4.4 INVESTIGATING THE COST-EFFECTIVENESS OF COMBINING TEST TECHNIQUES 96

4.5 EQUIVALENT MUTANTS' ANALYSIS 107

4.6 SURVEY DATA ANALYSIS 110

CHAPTER 5 QUALITATIVE ANALYSIS **113**

5.1 QUALITATIVE ANALYSIS OF LIVE MUTANTS 113

5.2 PROCEDURE TO IDENTIFY THE REASONS FOR NOT DETECTING FAULTS 114

5.3 CLASSIFICATION OF UNDETECTED FAULTS 116
5.4 INVESTIGATING THE VARIATION IN COST 122
5.5 ANALYSIS OF CODE COVERAGE 124
CHAPTER 6 THREATS TO VALIDITY 126
6.1 CONCLUSION VALIDITY 126
6.2 INTERNAL VALIDITY 127
6.3 CONSTRUCT VALIDITY 129
6.4 EXTERNAL VALIDITY 129
CHAPTER 7 PROPOSITIONS OF IMPROVEMENTS TO STATE TESTING 133
7.1 PROPOSITIONS FOR FAULT-DETECTION EFFECTIVENESS IMPROVEMENT 133
7.2 SUGGESTIONS FOR COST IMPROVEMENTS 144
CHAPTER 8 LESSONS LEARNED 153
CHAPTER 9 GUIDELINES ON THE USE OF STATE TESTING 159
CHAPTER 10 CONCLUSIONS 163
CHAPTER 11 REFERENCES 168
APPENDIX A CLASS CLUSTERS STATE MACHINES AND CLASS DIAGRAMS 177
APPENDIX B STATE INVARIANTS 191
APPENDIX C MUTATION OPERATORS 193
APPENDIX D PLOT OF MUTATION SCORE, NODE, EDGE AND RTP COVERAGE 194
APPENDIX E DESCRIPTIVE STATISTICS TABLES 198
APPENDIX F COMPLEMENTARINESS ANALYSIS DATA 201
APPENDIX G COST DISTRIBUTION AND CORRELATION WITH MUTATION SCORE 207
APPENDIX I MUTATION SCORES PER MUTATION OPERATOR 210
APPENDIX J ORDSET STATE MACHINE GUARD CONDITIONS 217
APPENDIX K CONTRACTS 218
APPENDIX L SURVEY QUESTIONNAIRES 229
APPENDIX M SURVEY DATA ANALYSIS 233

List of Tables

Table 1: Size of source code and state machines 17
Table 2: Research questions 22
Table 3: Distribution of experiment treatments among groups 31
Table 4: List of design and context changes across experiments in the series 36
Table 5: Participants and drivers distribution across experiments 38
Table 6: Summary of number of observations and mutants across experiments and clusters 39
Table 7: *t*-test results for the mutation score comparison 46
Table 8: *t*-test results for the mutation score comparison after removing trivial mutants 50
Table 9: Estimated effect size for 80% power 51
Table 10: *t*-test results for mutation score comparison differences across clusters 56
Table 11: t-test results for node coverage comparison across test techniques 62
Table 12: t-test results for edge coverage comparison across test techniques 63
Table 13: ANOVA - Impact of the test technique and its interaction with node coverage on mutation scores of OrdSet 67
Table 14: ANOVA - Impact of the test technique and its interaction with node coverage on mutation scores of Cruise Control 67
Table 15: ANOVA - Impact of the test technique and its interaction with node coverage on mutation scores of Elevator 67
Table 16: ANOVA - Impact of lab order and its interaction with test technique on mutation scores of Cruise Control 70
Table 17: ANOVA - Impact of lab order and its interaction with test technique on mutation scores of OrdSet 71
Table 18: ANOVA - Impact of lab order and its interaction with test technique on mutation scores of Elevator 71
Table 19: ANOVA - Impact of participant ability and its interaction with test technique on mutation scores of Cruise Control 72
Table 20: ANOVA - Impact of participant ability and its interaction with test technique on mutation scores of OrdSet 73

Table 21: ANOVA - Impact of participant ability and its interaction with test technique on mutation scores of Elevator 73
Table 22: ANOVA - Impact of cluster and its interaction with test technique on mutation scores. 74
Table 23: Drivers selection data for combining test techniques 77
Table 24: Combining test techniques - Paired *t*-tests results 85
Table 25: Testing differences across clusters 86
Table 26: Count of detected and live mutants per mutation operator (Carleton 1) 87
Table 27: Count of detected and live mutants per mutation operator (Sannio 1) 88
Table 28: Count of detected and live mutants per mutation operator (Carleton 2) 88
Table 29: Count of detected and live mutants per mutation operator (Sannio 2) 89
Table 30: Mean CPU time, LOC and number of method calls of collected drivers 92
Table 31: *t*-test results for cost analysis – CPU time 94
Table 32: *t*-test results for cost analysis - LOC 94
Table 33: *t*-test results for cost analysis - Number of method calls 95
Table 34: Comparing test techniques combination cost (Number of method calls) - Paired *t*-tests results 97
Table 35: Comparing test techniques combination cost (LOC) - Paired *t*-tests results 98
Table 36: Comparing test techniques combination cost (CPU execution time) - Paired *t*-tests results 98
Table 37: Observed increase in drivers' mutation scores and cost 100
Table 38: Mutation scores of augmented drivers 104
Table 39: Cost-effectiveness for augmented drivers with respect to state and structural drivers 105
Table 40: Count of live mutants per experiment and per treatment 108
Table 41: Equivalent mutants' impact on mutation scores 110
Table 42: Count of live mutants undetected by state drivers 114
Table 43: Distribution of categories of undetected faults among testing techniques 121
Table 44: Mutant count (percentage) per category of undetected faults and per cluster 122
Table 45: Classification of uncovered code 125
Table 46: Impact of proposed changes to state testing strategy on fault detection effectiveness of Cruise Control driver 141

Table 47: Impact of proposed changes to state testing strategy on fault detection effectiveness of OrdSet driver 142
Table 48: Impact of proposed changes to state testing strategy on fault detection effectiveness of Elevator driver 143
Table 49: Application of changes to improve drivers cost as number of method calls 148
Table 50: Impact of the application of propositions on LOC and CPU time of selected drivers 151
Table 51: Mutation operators' descriptions 193
Table 52: Mutation scores descriptive statistics 198
Table 53: Structural and state machine coverage descriptive statistics 199
Table 54: Cost descriptive statistics 200
Table 55: Mutation scores and difference fault sets scores (Carleton 1) 201
Table 56: Intersection score and ratios (Carleton 1) 201
Table 57: Union and not detected faults scores and ratios (Carleton 1) 202
Table 58: Mutation scores and difference fault sets scores (Sannio 1) 202
Table 59: Intersection score and ratios (Sannio 1) 203
Table 60: Union and not detected faults scores and ratios (Sannio 1) 203
Table 61: Mutation scores and difference fault sets scores (Carleton 2) 204
Table 62: Intersection score and ratios (Carleton 2) 204
Table 63: Union and not detected faults scores and ratios (Carleton 2) 205
Table 64: Mutation scores and difference fault sets scores (Sannio 2) 205
Table 65: Intersection score and ratios (Sannio 2) 206
Table 66: Union and not detected faults scores and ratios (Sannio 2) 206
Table 67: Model regression results of node coverage and test technique impact on mutation scores 209
Table 68: Mutation scores per mutation operator statistics for Cruise Control (Carleton 1) 210
Table 69: Mutation scores per mutation operator statistics for Cruise Control (Sannio 1) 211
Table 70: Mutation scores per mutation operator statistics for Cruise Control (Carleton 2) 211
Table 71: Mutation scores per mutation operator statistics for Cruise Control (Sannio 2) 212
Table 72: Mutation scores per mutation operator statistics for OrdSet (Carleton 1) 213
Table 73: Mutation scores per mutation operator statistics for OrdSet (Sannio 1) 214
Table 74: Mutation scores per mutation operator statistics for Elevator (Carleton 2) 215

Table 75: Mutation scores per mutation operator statistics for Elevator (Sannio 2).....216
Table 76: Survey Questionnaire for lab 1 (structural testing).....229
Table 77: Survey Questionnaire for lab 1 (state testing).....230
Table 78: Survey Questionnaire for lab 2 (state testing).....231
Table 79: Survey Questionnaire for lab 2 (sttructural testing).....232
Table 80: Comprehension and lab time availability by *Cluster* (Cruise Control & OrdSet).....233
Table 81: Comprehension and lab time availability by *Cluster* (Cruise Control & Elevator)...233
Table 82: Comprehension and lab time availability by *Lab*.....234
Table 83: Comprehension and lab time availability by *Ability*.....234
Table 84: Comprehension and lab time availability by *Experiment*.....234
Table 85: Test cases and oracles identification by *Cluster* (CruiseControl & OrdSet).....235
Table 86: Test cases and oracles identification by *Cluster* (CruiseControl & Elevator).....235
Table 87: Test cases and oracles identification by *Lab*.....236
Table 88: Test cases and oracles identification by *Ability*.....236
Table 89: Test cases and oracles identification by *Experiment*.....237
Table 90: Experience with tools by *Cluster* (Cruise Control & Elevator).....237
Table 91: Experience with tools by *Cluster* (Cruise Control & OrdSet).....237
Table 92: Experience with tools by *Lab*.....238
Table 93: Experience with tools by *Ability*.....238
Table 94: Experience with tools by *Experiment*.....238
Table 95: Time distribution among tasks by *Cluster*.....238
Table 96: Time distribution among tasks for all clusters, experiments and labs.....238

List of Figure

Figure 1: Mutant distributions 27
Figure 2: Cruise Control's drivers' mutation scores distribution 45
Figure 3: OrdSet's drivers' mutation scores distribution 45
Figure 4: Elevator's drivers' mutation scores distribution 45
Figure 5: Distributions of Carleton 1 mutation scores 47
Figure 6: Distributions of Sannio 1 mutation scores 47
Figure 7: Distributions of Carleton 2 mutation scores 48
Figure 8: Distributions of Sannio 2 mutation scores 48
Figure 9: Node coverage distribution of Cruise Control's drivers 59
Figure 10: Edge coverage distribution of Cruise Control's drivers 60
Figure 11: Node coverage distribution of OrSet drivers 60
Figure 12: Edge coverage ditribution of OrdSet drivers 60
Figure 13: Node coverage distribution of Elevator drivers 60
Figure 14: Edge coverage distribution of Elevator drivers 60
Figure 15: RTP coverage distribution for state drivers 65
Figure 16: Test technique and node coverage interaction effect graphs - Cruise Control 69
Figure 17: Test technique and Lab order interaction (Carleton 2 – Cruise Control) 72
Figure 18: Distribution of |Fc - Fs| / |F| % across experiments and clusters 79
Figure 19: Distribution of |Fs - Fc| / |F| % across experiments and clusters 80
Figure 20: Distributions of the commonly detected faults by drivers from both techniques 80
Figure 21: Distribution of |Fs ∩ Fc| / |Fs| ratio across experiments and clusters 81
Figure 22: Distribution of |Fs ∩ Fc| / |Fc| ratio across experiments and clusters 81
Figure 23: Distribution of |Fs ∪ Fc| / |F| ratio across experiments and clusters 83
Figure 24: Distribution of |Fs ∪ Fc| / |Fs| ratio across experiments and clusters 83
Figure 25: Distribution of |Fs ∪ Fc| / |Fc| ratio across experiments and clusters 84
Figure 26: Mutation scores for driver combination, structural and state drivers 101
Figure 27: Method calls for driver combination, structural and state drivers 101
Figure 28: Cost (LOC) for driver combination, structural and state drivers 102
Figure 29: Cruise Control - Running car activity diagram 137

Figure 30: Elevator's activity diagram for best elevator selection algorithm 139
Figure 31: Cruise Control's class diagram 177
Figure 32: Cruise Control state machine (Carleton 1 experiment) 177
Figure 33: Cruise Control state machine (Replication experiments) 178
Figure 34: Cruise Control's state machine mdeling real-time behavior 178
Figure 35: Cruise Control's transition tree (Carleton 1 experiment) 179
Figure 36: Breadth-first traversal Cruise Control's transition tree 179
Figure 37: Depth-first search traversal Cruise Control's transition tree 180
Figure 38: OrdSet's class diagram 181
Figure 39: OrdSet's state machine 182
Figure 40: Breadth-first search traversal OrdSet's transition tree 184
Figure 41: Depth-first search traversal OrdSet's transition tree 185
Figure 42: Elevator's class diagram 186
Figure 43: Elevator's state machine 187
Figure 44: Breadth-first search traversal Elevator's transition tree 189
Figure 45: Depth-first search traversal Elevator's transition tree 190
Figure 46: Node and Edge coverage distributions (Cruise Control – Carleton 1) 194
Figure 47: RTP coverage distribution (Cruise Control – Carleton 1) 194
Figure 48: Node and Edge coverage distributions (OrdSet – Carleton 1) 194
Figure 49: RTP coverage distribution (OrdSet – Carleton 1) 194
Figure 50: Node and Edge coverage distributions (Cruise Control – Sannio 1) 195
Figure 51: RTP coverage distribution (Cruise Control – Sannio 1) 195
Figure 52: Node and Edge coverage distributions (OrdSet – Sannio 1) 195
Figure 53: RTP coverage distribution (OrdSet – Sannio 1) 195
Figure 54: Node and Edge coverage distributions (Cruise Control – Carleton 2) 196
Figure 55: RTP coverage distribution (Cruise Control – Carleton 2) 196
Figure 56: Node and Edge coverage distributions (Elevator – Carleton 2) 196
Figure 57: RTP coverage distribution (Elevator – Carleton 2) 196
Figure 58: Node and Edge coverage distributions (Cruise Control – Sannio 2) 197
Figure 59: RTP coverage distribution (Cruise Control – Sannio 2) 197
Figure 60: RTP coverage distribution (Elevator – Sannio 2) 197

Figure 61: RTP coverage distribution (Elevator – Sannio 2) 197
Figure 62: Cost comparison of state and structural drivers (Carleton 1) 207
Figure 63: Cost comparison of state and structural drivers (Sannio 1) 207
Figure 64: Cost comparison of state and structural drivers (Carleton 2) 207
Figure 65: Cost comparison of state and structural drivers (Sannio 2) 208

Chapter 1

INTRODUCTION

There is an increasing interest [55] in model-driven development for object-oriented systems, using for example the Unified Modeling Language (UML). In addition to be a key resource for designing object-oriented software and providing means for communicating ideas among designers and customers, models are very useful to guide and automate the testing of object-oriented software. In particular, UML state machines are very useful to model the most complex and critical components in object-oriented software [46].

Other than being useful to support development, models can be used to support testing activities. Model-based testing has been assessed in a number of empirical studies and showed to be useful in systematically defining test strategies and criteria, and deriving test cases and oracles [20, 24, 25, 31, 74, 76]. At the same time, a number of researchers conducted studies on the cost and effectiveness of conventional testing strategies: white-box [42, 43, 53, 91] and black-box testing strategies [20, 78, 92]. However, despite a growing number of studies [18, 20, 21, 24, 25, 31, 73, 74], more empirical evidence is required to assess the importance of models in improving testing cost-effectiveness, and to investigate how model-based testing can be combined or augmented with simpler, widely adopted testing techniques such as white-box structural testing. Therefore,

assessing the cost and fault detection effectiveness of testing techniques based on state machines and comparing them with simpler, code coverage-based techniques seems a logical investigation to undertake. The latter being a basic test practice automated by existing commercial tools, only a significant improvement in fault detection effectiveness or cost would justify the use of approaches based on state machines.

This thesis focuses on the effectiveness of UML state machine-based testing when compared and augmented with white-box, structural testing, which is deemed to be the most common basic technique for testing (clusters of) components. At the same time, the most complex and critical components in object-oriented software are also the ones which, according to mainstream UML development methods, would likely be modeled with state machines [46]. Therefore, assessing the cost and effectiveness of testing techniques based on state machines and comparing them with simpler, code coverage-based techniques seems a logical investigation to undertake. The latter being a basic test practice automated by existing commercial tools, only a significant improvement in fault detection effectiveness or cost would justify the use of approaches based on state machines. The choice of UML as a language to model state machines is a practical one, as UML is becoming a widely known and applied standard in industry. Specifically this thesis focuses on a specific state machine-based test strategy, i.e., round-trip paths testing [16], an adaptation of the W-method [32] for UML state machines. The choice is driven by previous work on the subject [20], showing that this is a reasonable compromise in terms of fault detection effectiveness and cost between cheap but inefficient criteria (e.g., all transitions) and efficient but expensive criteria (e.g., all transition pairs).

In this thesis, based on controlled experiments involving trained participants, we perform both a quantitative analysis of differences in fault detection effectiveness and cost among test

techniques, and a qualitative analysis to understand the reasons for these differences and the variations observed across test drivers and component clusters. The thesis aim is to answer the following research questions:

- How does the fault detection effectiveness of test cases identified and manually generated by testers based on the state machine alone (functional testing) compare to that of test cases manually generated by testers based on the coverage analysis of source code (structural testing)?
- Are the sets of faults detected by state machine-based testing and structural testing techniques complementary? Does this suggest that somehow combining the two techniques is beneficial in terms of fault detection effectiveness? Given that test cases derived from state machines are available earlier, can state machine testing be effectively augmented, based on its code coverage analysis, with structural test cases?
- How does code coverage in terms of node and edge coverage in the cluster under test achieved by state machine testing compare to that of structural testing?
- How does the trade-off between cost and effectiveness (cost-effectiveness) compare for state machine-based testing and structural testing?
- What are the different factors that impact the effectiveness of state machine-based testing techniques?
- More specifically, how can the state machine-based, round-trip testing strategy be modified to improve its fault-detection effectiveness and lower its cost?

As no analytical means can help us obtain realistic, practically useful answers to these questions, a series of four controlled experiments were conducted —involving fully trained, undergraduate

and graduate students—on three object-oriented class clusters[1] with a non-trivial state-dependent behavior modeled using UML state machines. Results show that:

- Overall, techniques based on code coverage and state machines do not show practically significant differences in terms of fault detection effectiveness.
- The two techniques are, to a significant extent, complementary in terms of the faults they detect. When augmenting state machine testing with structural testing, significant improvements in fault detection can be observed.
- The real-time behavior of a class cluster negatively affects the effectiveness of both structural and state-based testing. This suggests that the techniques used in the experiments need to be complemented with testing techniques specifically targeting real-time properties.
- A number of improvements can be applied to the round-trip paths testing strategy to improve both its fault detection effectiveness and its cost-effectiveness.

The thesis also presents some guidelines for testers based on the lessons learned from this study on when to use state machine-based testing and how to integrate it with other test strategies.

The thesis is organized as follows: Chapter 2 discusses the related literature and Chapter 3 provides a detailed description of the conducted controlled experiments while Chapter 4 presents and analyses the results. Chapter 5 details the outcome of a qualitative analysis to understand the limitations of the state machine testing technique, and the factors affecting the cost of developing test drivers. Threats to validity are discussed in Chapter 6. Based on the results of the experiments and the observations from their executions, a number of changes to the state

[1] In the context of object-oriented unit testing, a class cluster is a set of related classes constituting a software system unit.

machine testing strategy are proposed in Chapter 7 to improve its fault detection and cost-effectiveness. Chapter 8 summarizes the lessons learned from conducting the experiments and lists guidelines for the design and the execution of experiments in software testing. Guidelines on where and how to use state machine-based testing are presented in Chapter 9. Finally, Chapter 10 concludes and summarizes the results.

Chapter 2

RELATED WORK

As discussed above, we address the fault detection and cost effectiveness of a state-based testing technique, the round-trip path strategy [16], by empirically comparing it with and combining it to traditional structural testing. We will therefore restrict the discussion of related work to state-based and structural testing (Section 2.1) with a focus on empirical studies aimed at assessing the usefulness and the effectiveness of testing techniques (Section 2.2).

2.1 State-Based and Structural Testing

One of the earliest works on state-based testing is the work by Chow [32] who proposed the W-method for finite state machines (FSM). This method has been adapted to UML state machines by Binder [16] and renamed the round-trip paths (RTP) strategy. In both techniques, the state model is traversed to 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. When there are guard conditions on transitions and these guards are in a disjunctive form then several transitions are warranted, one for each truth value combination that is sufficient to make the guard condition true.

Other state-based test crieria were proposed by Offutt et al. [76]: transition coverage, full predicate coverage, transition-pair coverage, and complete sequence. A case study was used to compare these criteria with a random selection of test cases. Results showed an improvement in fault detection when using the full-predicate coverage criterion, while transition coverage yielded a smaller number of test cases than random testing, with the same fault detection rate and branch coverage (of the source code control flow graph).

Additional testing strategies have been defined for state machines. Hong et al. [51] propose a technique to derive extended finite state machines (EFSM) from state machines. The EFSM is then transformed into a flow graph modeling the control flow and data flow in the state machine thus enabling the application of conventional control and data flow analysis techniques. A modification of this method is described by Bogdanov and Holcombe [18] to address the compliance of an implementation of a system to its specification. This has been further extended to UML state machines with operations contracts defined in the Object Constraint Language (OCL) [23].

FSMs, EFSMs, and UML state machines have been widely used to model systems in various application domains, such as sequential circuits and communication protocols. The study in [58] provides a rather complete and clear review of the fundamental problems in finite state machines testing.

One problem of special interest to us is conformance testing [58], which is to test whether the implementation conforms to the specification. All the proposed methods for conformance state-based testing have the same basic objective: To ensure that every transition of the specification state machine is correctly implemented. The methods however differ in terms of the types of sequences of inputs they use to verify that the machine is in the right final state. For instance, one

strategy consists in constructing a breadth-first-traversal transition tree and checking states in that order. For every state, a checking sequence is applied to verify if the system under test is in the expected destination state [58]. Similarities can be seen between this strategy and the RTP technique. Both strategies are based on the generation of a tree from the state machine in which the paths from the root state along the branches of the tree correspond to test cases aimed at comparing a state machine to its implementation. One difference between the two strategies is the method for identifying visited states: The RTP technique uses state invariants to differentiate states and the other strategy uses separating sequences[2] for this purpose.

One important application of conformance testing is protocol testing where FSMs define protocol specifications [17]. Many systematic methods were defined for testing protocol implementations against their specifications. Some of these methods are guided by heuristics [13, 72]; others are based on fault models [61, 80, 94]. These methods determine distinguishable states traversed by a test sequence with the aim of verifying the conformance relationship between the protocol specification and its implementation. As opposed to other finite state machine testing strategies, protocol conformance testing strategies mainly address non-deterministic finite state machines [17].

It is worth noting that any of the criterion discussed in the literature, including the W-method, specifies a test case as a (sub-)sequence of transitions and states in the state machine. Though deriving those test case specifications can easily be automated, due to possibly complex guard conditions, many sequences may turn out to be infeasible and identifying test (input) data for these test case specifications is in practice very difficult [22]. This problem is identical to the sensitization problem described by Beizer [12] for white-box testing, unless the guard conditions

[2] Let $\lambda(s_i, x)$ be the output function which is the result of applying the sequence of inputs x to the state s_i. A sequence x is said to be a separating sequence for state s_i when $\lambda(s_i, x) \neq \lambda(s_j, x)$ for all state $s_j \neq s_i$.

are simple (see for instance [76]). This sensitization problem is un-decidable, which explains why some approaches, based on search-based optimization techniques, have been recently developed to derive test cases for object oriented systems (e.g., [67]). For this reason, in this thesis we will focus on the assessment of state-based testing (and its comparison with structural, white box testing) when manually performed by testers.

2.2 Empirical Studies

From a more general standpoint, a growing number of empirical studies address the cost effectiveness of testing strategies in various types of testing techniques: white-box [42, 43, 91], black-box [92], or model-based [8, 20, 25, 73, 82]. Many of these studies use the mutation strategy to seed faults and evaluate the fault detection effectiveness of the testing techniques. For instance, a simulation and analysis procedure [25] has been proposed and used to study the cost-effectiveness of four state machine-based coverage criteria, namely all-transitions, all-transition-pairs, full-predicate [76], and round-trip paths [16]. Briand *et al.* [25] showed that the cost effectiveness of these criteria depends to a significant extent on the characteristics of the state machine. For state machines labeled with numerous guard conditions, the round-trip paths strategy provides a good compromise between all-transitions and all-transition-pairs, the latter being far too expensive and the former rather ineffective.

An empirical study focusing on white-box testing strategies was performed by Frankl and Weiss [42] where the all-uses and decision (all-edges) criteria were compared to each other and to the null criterion (random test suites). Results showed that all-uses was not always more effective than the decision and the null criteria but in few cases where there was a difference, the difference was, however, large. In contrast, in cases where the decision criterion was more effective than the null criterion, the difference was small. The results of Frankl and Weiss were

further confirmed by Hutchins *et al.* [53], whose experiments showed a better fault detection effectiveness of the all-uses criterion over the all-edges criterion, to the expense of larger test sets. The two techniques seem to be complementary in terms of the faults detected. This is yet another example of the benefit of combining different testing techniques on the overall fault detection effectiveness of test sets.

Pretschner et al. focused on the automatic generation of test cases on the grounds of symbolic execution with Constraint Logic Programming (CLP) [82]. The aim of the symbolic execution of the model is to find an execution trace—and therefore a test case—that leads to the state to be tested. A number of strategies are used to optimize the traversal of the state machine. For instance, a fitness function is defined to generate the shortest path to the destination state. Other strategies include attributing probalilities to transitions or storing visited states and transitions to prevent repeated visits. The study aims at comparing the use of behavioral models namely extended finite state machines (EFSM) to hand-crafted tests generated based on requirements' message sequence charts (MSC). Results show that the use of models significantly increases the number of detected requirement errors. However, the number of detected programming errors was unrelated to the use of models. Hand-crafted, model-based tests showed to detect as many errors as automatically generated tests, but the two sets of tests detect partially different sets of faults [82]. Our work differs in that it compares a more widely applied state machine testing technique (RTP) to a practically common and widely used structural testing technique (nodes and edge coverage). It does so by performing replicated, controlled experiments involving human participants that aim at precisely understanding the limits of each technique and how they complement each other.

An approach in model-based testing was proposed and validated by Nebut *et al.* in [73]. The approach consists in automating the generation of system test scenarios from use cases in the context of object-oriented embedded software. By using contracts with UML use cases, the authors apply Meyer's Design By Contract approach [68] at the requirement level. Executable contracts written in terms of logical expressions allow for defining valid sequences of use cases and extracting relevant paths which are called test objectives. Subsequently, test scenarios are generated from test objectives. An empirical evaluation of the proposed approach was executed on three small case studies (800 LOC to 2000 LOC) to assess the efficiency of the generated test cases in terms of statement coverage. A classification of the code under test aimed at identifying functional code and differentiating it from other code categories (dead code and robusteness code). The results showed that most of the functional statements in the code are covered by the proposed technique with a relatively small set of test cases.

Briand *et al.* [20] focused on the cost effectiveness of the RTP technique [16]. They investigate, in controlled experiment settings, the fault detection effectiveness of state-based (RTP) testing for classes or class clusters modeled with state machines. They also investigate how to augment RTP with a well-known black-box testing technique: category-partition (CP) [78]; though this part of the study is very limited. The study was based on a series of controlled experiments where the RTP technique was applied on a number of systems with two different levels of oracle precision. Drivers implementing the RTP technique were then augmented with an implementation of the CP technique [78]. Though useful at detecting faults, results showed that the RTP technique needed to be complemented with CP to significantly increase its fault detection. These results were one of the motivations for our study. The cost of applying the category-partition technique is high and the human resources available for applying it may be

limited. However, a code coverage, structural technique can be easily applied. By using code coverage instrumentation, structural testing can be helpful at identifying those parts of the cluster under test that were not tested by the state-based testing technique. The limitations of the RTP technique incited us to further identify the factors that affect its fault-detection effectiveness. However, in contrast with Briand *et al.* [20], we choose to compare and complement the RTP technique with a structural testing technique rather than with a black-box testing technique. Furthermore, whereas CP was applied to augment RTP on a small subset of methods, here structural testing is fully applied to the same extent as RTP, compared, and combined. It is expected that structural testing can be helpful at better exercising those parts of the cluster under test that were not (sufficiently) tested by state-based testing, and that can be identified by analyzing its code coverage. This is of pratical importance as state models are rarely complete and fully defined in practice. Briand *et al.*[20] also noticed the significant difference in terms of fault detection and cost between two oracle strategies, one using precise oracles checking the concrete state of objects (i.e., checking all attribute values), and the other is based on state invariants. We also address this issue and suggest ways to limit the cost of oracles without affecting their fault detection effectiveness.

A growing number of studies [11, 26, 60, 69, 83, 84] have been discussing the importance of replicating empirical studies to increase the credibility and the generality of empirical results. A study may be replicated externally or internally[3], with or without experiment design changes. In this research study, we replicated three times an original experiment across two geographical locations, with slight changes in the experiment plan to address some uncovered threats to

[3] Replication takes two forms: internal and external. Internal replication is undertaken by the original experimenters while external replication is undertaken by independent researchers and is critical for establishing sound results [34].

validity and to overcome some of the limitations encountered in the original experiment. More details on the changes and their rationale are provided in Chapter 3.

Basili *et al.* [11] discussed the importance of having families of experiments on building credible knowledge in software engineering through empirical studies. These experiments would have in common the same research questions or sometimes extend the studied theory by varying research questions to investigate other aspects related to the study. Families of experiments are of great value for building knowledge around existing and new software technologies and processes. As an example, Basili *et al.* [11] reflect on a set of software reading techniques' experiments and propose an organizational framework for experiments. The objective of the framework is to provide support for experimenters on how to better define experiments and combine them to overcome validity problems. It addresses the modeling of processes and their effectiveness, as well as context modeling and consequences of experimental designs on threats to validity. The proposed framework supports families of experiments and facilitates their abstraction by building knowledge on top of them. This framework is based on the GQM (Goal / Question / Metric) template. In their study, the authors identified different types of replications varying from strict replications to those varying research questions or even extending the studied theory. They put strong emphasis on lab package design. The cost of an experiment increases greatly because of the preparation of the different required artifacts. Thus reusing artifacts can reduce experimentation cost. Lab packaging then helps reduce such cost and facilitates the experimenter's work in experiments' preparation.

Chapter 3

EXPERIMENT DESCRIPTION

This section reports, following a specific template [93], the definition and planning of the experiments we performed. First, we define the objective of the original experiment and its context (Section 3.1), and then we describe the plan of the experiment including the context selection criteria, the research questions, and the experiment design (Section 3.2). In Section 3.3 we describe how the experiments were prepared and executed. Section 3.4 describes the differences between the first experiment and its subsequent three replications, and the rationale for the changes that were made to the experiment design and operation.

3.1 Experiment definition and context

The goal of this study is to analyze *the state machine based round-trip path testing strategy and the edge coverage structural testing technique* for the purpose of *comparing and assessing them as well as their combination* with respect to *their fault detection effectiveness and cost effectiveness* from the point of view of *the tester. The* context *consists of* objects, *i.e., source code and UML state machines of three Java software clusters, and* participants, *i.e., undergraduate*

students from the 4th year of software engineering at Carleton University, Canada, and graduate students from the Master in Software Technology of the University of Sannio, Italy.

The study is conducted as a series of four experiments: two conducted at Carleton University, Canada, and two at the University of Sannio, Italy; since replication is imperative to increase the credibility of results and to allow more robust conclusions to be drawn.

This comparison of state-based testing and structural testing techniques we conducted is of practical importance for a number of reasons. First, various forms of state-based testing have been discussed for many years in the research literature, whether for software components or protocols (Chapter 2). Despite this, we know very little about the benefits of such practice for software testing. The focus on UML state machines is motivated by practical reasons, as we want to place our work in the context of UML-based development. The comparison with structural testing is due to its wide application in practice, at least in its simplest forms, and it can therefore be considered a reasonable baseline of comparison. Such a comparison, however, only makes sense when testing software components (e.g., classes, clusters) that have a state-driven behavior. Though such components are far to represent the largest proportion of classes in a typical cluster, the most complex components are usually state-driven.

When referring to state machine models, we do not include only the state machines themselves but also the related artifacts that are required to understand them: class diagrams, class public interfaces (signatures, attributes), contracts and state invariants, and a textual, high-level description of the software functionalities and objectives. However, as participants working with the UML artifacts are expected to use the state machine diagram to generate test cases, for the sake of brevity, we will simply refer to them as a "state machine model" in the remainder of the thesis.

The experiments involved three Java class clusters, that all have a state-driven behavior:

a. `OrdSet` is a Java class (of 393 lines of code – LOC) that was included in the original experiment and its first replication. Each instance of `OrdSet` represents a bounded, ordered set of integers. The `OrdSet` class provides methods for adding a single element, removing a single element, and creating the union of two ordered sets.

b. `Cruise Control` is a cluster of four Java classes (of 358 LOC). It simulates a car engine and its cruising controller.

c. `Elevator` is a cluster of eight Java classes (of 581 LOC) that was included in the last two replications only, as a way to make our results less dependent on the first two clusters. It consists of a number of elevators servicing a number of floors. An elevator accepts stop requests to travel to other floors. All elevators start at the first floor. Users can also request service from floors to go up or down.

The above three class clusters were extracted from a pool of software engineering students' final year projects, where teams of students follow a rigorous, UML-based, development strategy. Requirements of the final projects were identified and described in use case diagrams. Other UML artifacts were also used to model the systems, including class diagrams, collaboration diagrams, activity diagrams and state machine diagrams. Since these students were carefully trained for four years, we expected their models and implementations to be representative of the best we could expect in practice. These implementations and corresponding state machines were simplified from their original versions implemented by the engineering students in order to give testers sufficient time for testing them within the duration of laboratory sessions. These class clusters were selected in part because of their differences. They represent two typical cases where

a state machine is used to model the behavior of a complex data structure (`OrdSet`) and a state-dependent control class in a real-time multithreaded control cluster (`Cruise Control` and `Elevator`). Source code and models of the three clusters can be found on the Software-artifact Infrastructure Repository (SIR) [6].

The three class clusters are of varying code and state machine complexities, which we summarize in Table 1. For each cluster, the table reports the number of classes in the cluster, the total number of operations and attributes, the cluster size in lines of code (LOC) corresponding to the sum of classes' LOC measures, the number of control-flow statements, i.e. if/else, for and while statements, the total number of nodes and edges in the cluster's control flow graph, and the number of transitions, states and events in the state machine. The numbers in Table 1 correspond to the simplified versions of the clusters used in the series of experiments.

	Cruise Control	OrdSet	Elevator
# classes	4	1	8
# operations	34	23	74
# attributes	14	5	37
# LOC	358	393	581
# control flow statements	33	36	56
# nodes	106	111	241
# edges	103	126	214
# transitions	17	22	50
# states	5	5	6
# events	7	5	10
# guard conditions	0	17	19

Table 1: Size of source code and state machines

Though `OrdSet` is composed only of one class, one can note that its state machine and control flow are more complex than those of `Cruise Control`; this is visible from the number of control flow edges in the `OrdSet` source code and the number of transitions in its state diagram. Furthermore, the guard conditions in the `OrdSet` state machine add to the complexity of the

class, whereas `Cruise Control` is event-driven only. `Elevator` is far more complex than the two other clusters. Both its code size attributes such as LOC and number of nodes and its state machine complexity attributes such as number of transitions and number of events are greater than those of the two other clusters.

The choice of these three clusters was in part based on the fact that each state machine was created at a different level of abstraction. While the `Cruise Control`'s state machine is simple (no guard conditions, nor complex actions), it is restricted to the state behavior of the cluster without modeling its real-time behavior. The real-time behavior is implemented in two algorithms in the code. The first represents the relation between time and class attributes such as speed and distance. The second implements a relation between car throttle, time and cruise control speed to control car speed while in cruising state. The state machine therefore models the `Cruise Control`'s behavior at a high level of abstraction without modeling algorithms managing the cluster's real-time behavior and speed control. On the other hand, the `OrdSet`'s state machine models the different functionalities of the system at a relatively low level of abstraction: all the functionalities (algorithms) are specified under the form of states, transitions, guards, and actions (pre and post conditions). The third cluster, `Elevator`, has a rather complex state machine at a very low level of abstraction: its real-time behavior is much simpler than `Cruise Control`'s and can be partly specified in the state machine; the `Elevator`'s state machine models the `Elevator` class state-dependent behavior, the only class in the cluster that exhibits a state-dependent behavior. By exercizing the state machine, the interactions between the classes in the cluster are indirectly exercised. The state machine does not model all of `Elevator` functionalities, for instance avoiding the specification of interactions between different instances of elevators (e.g., optimization algorithm for choosing the best elevator to

service a floor) and not differentiating between moving directions (up and down). In our experience, such variations in levels of abstraction are common in modeling practice as a trade-off between modeling cost and completeness must be achieved.

The source code used in these experiments is admittedly small. However, it is important to note that state machines, in UML-based development, are mostly used to model the behavior of complex classes or class clusters [27, 46, 57], particularly complex data structures (usually referred to as entity classes) and control classes, for example in reactive systems. They are rarely used to model entire systems, as this would result in large and unmanageable models for software engineers and testers, or in simplistic, incomplete models, which do not adequately capture the system behavior. Furthermore, even when the source code is small, the state-behavior can be quite complex when measured in terms of states, events, and transitions. This issue is further discussed in Chapter 5.

3.2 Experiment planning

There are many ways we could have designed, planned, and executed experiments to achieve our objectives. Before delving into details in the next subsections, let us provide a high-level view of the experimental approach we adopted and its rationale. In an experimental, artificial setting, the time allocated to an experiment is necessarily limited. What this implies is that test techniques will be compared assuming a limited, equal effort on the part of testers. From a comparison standpoint, this is fair but in terms of absolute fault detection effectiveness, the results could, in practice, look very different depending on the effort involved. Note that in practice the time dedicated to testing is also constrained and we therefore do not consider this as a limitation given our objectives.

Another important point to highlight, which is further discussed in [19], is that techniques that involve humans can, of course, be incorrectly or incompletely applied, especially when under time constraints. The question is then whether we want to assess a technique in terms of its maximum potential, when perfectly implemented, or whether we want to account for human factors. In our context, we chose the latter, thus justifying the use of experimentation with human participants. Furthermore, to be realistic in terms of human factors, we need to ensure we provide testers with adequate support that is representative of what could be expected in the current state of technology. For instance, participants would be trained for the tasks in laboratories previous to the experiment. They would also be provided with driver templates.

This section details the experimental plan, describing the context, the research questions, the variables, and the design.

3.2.1 Context selection

The participants involved in the four experiments had the following characteristics:

- First and third experiment (Carleton 1 and Carleton 2): participants are fourth-year students from a specialized, software engineering bachelor program. They were well versed in Java and UML and were attending a course on software testing that covers different white-box and black-box testing techniques with a focus on object-oriented testing. The experiments were conducted during the lab hours of that course as part of practical lab exercises. 48 students participated to the first experiment, and 19 to the third.
- Second and fourth experiment (Sannio 1 and Sannio 2): participants are graduate students attending a Master in software technology. Master participants (about 30 every year) are selected from a population of 300 graduate students in computer science and computer

engineering. The participants were students attending an intensive course on software testing. Their experience with software testing before attending the course varied from no experience to some experience with JUnit. 25 students participated to the second experiment, and 19 to the fourth one.

The method for the selection of participants follows a stratified random sampling[4]; participants were first assigned to blocks based, for Carleton Experiments, on their background and knowledge of object-oriented design and development techniques[5], and for Sannio experiments on *laurea* graduation score (since participants were graduate students). Then, participants were randomly selected from the different blocks to form four groups with a similar distribution to ensure the results would not be affected by random variations in participant experience across groups. In addition, groups were defined to be of similar sizes to ensure a balanced contribution of test techniques/clusters combinations to the results. However, there were practical constraints regarding the availability of certain participants and this limited the randomization of selection. In spite of this issue, we managed to ensure that we had comparable block distributions across groups where each block is represented by a similar number of participants in every group.

3.2.2 Research questions

In this section we provide a detailed description of the research questions to be addressed by the experiments (Table 2) to address the objectives listed in Section 3.1.

---

[4] Stratified random sampling: The population is divided into a number of groups or strata with a known distribution between the groups. Random sampling is then applied within the strata [59].

[5] The background and knowledge regarding object-oriented design and development techniques of the different subjects was measured based on their grades in two advanced courses in software engineering and object-oriented design.

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

Table 2: Research questions

The fault detection effectiveness of both state machine-based and structural test techniques is addressed in research question RQ1. In RQ2, we compare the code coverage level of drivers of both state machine-based and structural test techniques. RQ4 investigates how complementary the two techniques are in terms of fault detection. Answering RQ3 and RQ5 allows us to identify factors that have an interaction effect with the test technique on fault detection effectiveness. In RQ6 the goal is to identify fault types for which test techniques are a better detector, thus helping us identify conditions under which state machine-based testing is adequate. Answers to RQ7 are used to compare test techniques with respect to their cost, both individually and when combined. RQ8 focuses on assessing the cost-effectiveness of augmenting state testing with structural

testing. The reason why we do not investigate the reverse option, i.e., augmenting structural testing with state testing, is that state machine test cases can be derived earlier from design models and are therefore available when the source code becomes available. In addition, we will investigate the reasons why test techniques differ across class clusters in terms of fault detection and cost. With RQ9 we investigate ways of improving state machine-based testing to improve its fault detection effectiveness.

3.2.3 Variable selection

At a high level, our dependent variables are based on the following constructs: (1) Fault detection effectiveness, overall and across different fault types; (2) Cost for both test specification and execution. There is one independent variable of interest (treatment): The type of artifacts used as a basis for testing (i.e., state machine model or code structure). Also, as further discussed below, a number of other variables (co-factors) are accounted for to determine whether they interact with the effect of our independent variable: code coverage, learning effects, participant ability.

For the sake of brevity, we will refer to state machine-based testing (drivers) using the round-trip path strategy as state testing (drivers). The same applies to edge coverage structural testing (drivers) which is simply referred to as structural testing (drivers).

To address the research questions listed in Table 2, the high-level dependent variables above mentioned were further detailed as follows:

1. The faults (mutants) detected using state machines (*Fs*) and source code (*Fc*) among all the faults (F[6]). The purpose here is to compare the effectiveness of state testing and structural testing in terms of their fault detection capability (RQ1).

[6] $Fs \subseteq F$, $Fc \subseteq F$, and $Fs \cup Fc \subseteq F$ (state machine and source code testing, combined, may not find all the faults).

2. The faults detected by both test techniques ($Fs \cap Fc$). This is a measure of how redundant the two techniques are (RQ4).

3. The faults detected only by state testing ($Fs - Fc$). We can thus evaluate the effectiveness of state testing to detect faults that are not detected by structural drivers (RQ4).

4. The faults detected only by structural driver ($Fc - Fs$). This helps us to identify the weaknesses and limitations of state testing (RQ4).

5. The ratios $|Fc - Fs| / |Fc|$, $|Fc - Fs| / |Fs|$, $|Fs - Fc| / |Fs|$, $|Fs - Fc| / |Fc|$, $|Fs \cap Fc| / |Fs|$, $|Fs \cap Fc| / |Fc|$ (RQ4). These ratios represent different ways to measure the extent to which each type of driver complements the other type. For instance, if $|Fc - Fs| / |Fc| = 0.7$ then 70% of the faults detected by the structural driver are not detected by the state driver. This portion of detected faults represents the contribution of the structural driver in complementing state driver to achieve a more effective testing strategy.

6. The faults detected when combining state and structural test cases ($Fs \cup Fc$). The purpose here is to evaluate the effectiveness of combining techniques to overcome their individual limitations (RQ4).

7. The ratios $|Fs \cup Fc| / |Fc|$ and $|Fs \cup Fc| / |Fs|$. The purpose here is to evaluate the relative improvement in fault detection resulting from combining test techniques (RQ4).

8. The same variables as in 3, 4, and 6 but for each specific type of fault, i.e., mutation operator in our context. The purpose here is to answer the above questions for each mutation operator and analyze differences (RQ6).

9. The number of calls in test drivers to methods in classes under test (*MC*). The purpose here is to evaluate and compare the cost of testing strategies using a surrogate measure for test driver execution cost (see Section 3.2.6) (RQ7).

10. The CPU execution time of test drivers to methods in milliseconds. The purpose here is to evaluate and compare the cost of testing strategies using a surrogate measure for test driver execution time (see Section 3.2.6) (RQ7).

11. The number of lines of code in test drivers (*LOC*). The purpose here is to evaluate and compare the cost of testing strategies using a surrogate test driver size measure (see Section 3.2.6) (RQ7).

12. For each test technique, cost-effectiveness is computed as the ratio of faults detected (*Fs* or *Fc*) over the cost (RQ8).

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

3.2.4 Mutant seeding

To evaluate the fault detection effectiveness of the experimented techniques, we executed the different drivers delivered by the experiment participants on a number of mutant programs (or mutants), that is versions of the program under test where one fault was seeded using a mutation operator [62, 63], and used different kinds of oracles (Section 3.2.5) to compare the behavior of the original program with those of mutants. Whenever the oracle indicates a difference, the mutant is considered killed. The fault detection effectiveness of a driver is measured by the

percentage of mutants killed. The mutants are automatically generated using MuJava [64], which allows to quickly generate a large number of faults thus facilitating statistical data analysis. MuJava uses two types of mutation operators, class level and traditional method level operators. The main motivations for using MuJava is to apply a systematic, automated, and independent mechanism to generate a large number of faults thus facilitating statistical data analysis [7]. Threats to validity related to the use of mutation operators are discussed in Chapter 6.

Table 51 in Appendix C includes the list of all mutation operators used in this experiment along with a brief description. Figure 1 shows the distribution of the created mutants among the different mutation operators for the three clusters. The distributions look different due to differing code characteristics. For example, eight mutants have been created with the AORS mutation operator (Arithmetic Operator Replacement – shortcut) for `OrdSet`, six for `Elevator`, and none for `Cruise Control` which has no shortcut arithmetic operators, i.e., ++ and --; 12 mutants have been created with the ASRS mutation operator (Assignment Operator Replacement – shortcut) for `Cruise Control`, and none for `OrdSet` which has no shortcut assignment operator, i.e., +=, -=, /=, *= and %=. A large number of AOIS mutants (Arithmetic Operator Insertion – Short-cut) have been created for `Elevator` (556 vs. 297 for `OrdSet` and 144 for `Cruise Control`), since `Elevator` includes a larger number of computation statements. The large number of AOIS mutants compared to other types of mutants can be associated to the fact that the AOIS mutation operator consists of adding the arithmetic operators ++ or -- to numerical attributes and parameters. Therefore, for each occurrence of a numerical attribute or parameter, four mutants are created (one right and one left assignment for each of the two operators).

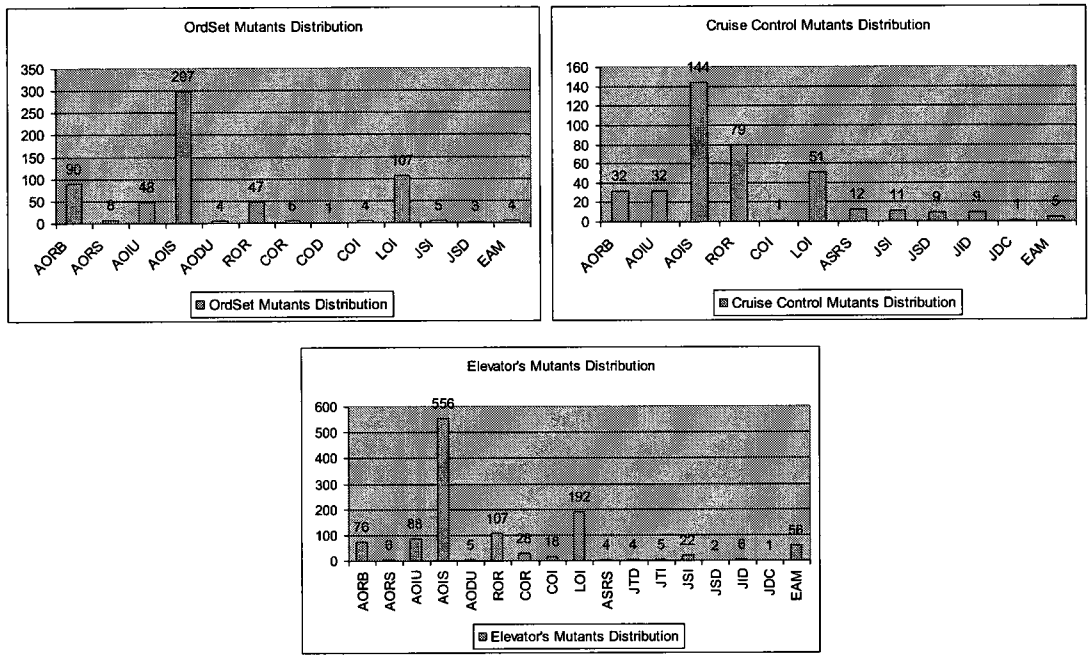


Figure 1: Mutant distributions

One issue to be addressed is the detection of equivalent mutants, i.e., mutants that have the same behavior as the original program and therefore cannot be killed by test cases. Manually identifying equivalent mutants is the most common practice but is time-consuming and error-prone. A number of studies addressed this issue and proposed optimization techniques to automate the detection of equivalent mutants [75, 77]. However, these methods have shown to detect on average only half of the equivalent mutants, and the detection ratio depends heavily on the program characteristics [77]. Instead, some authors proposed, as a heuristic, to consider live mutants not killed by any test case in the overall test pool as equivalent mutants [10, 28, 36]. This approximation is thought to be good enough especially when dealing with a large number of mutants. Also, the exclusion of these mutants does not influence the comparison of different testing strategies (i.e., black-box state-based testing vs. white-box code-based testing) since such mutants are not considered in both cases.

However, in our study we cannot simply assume that mutants not detected by any driver are equivalent as we know the testing performed by our experiment participants is incomplete and unlikely to kill all non-equivalent mutants. Therefore, we do not attempt to discard non-detected mutants as equivalent mutants but present our results based on all mutants and then perform a manual, qualitative analysis of all undetected faults to assess the potential impact of equivalent mutants on the fault detection effectiveness results.

3.2.5 Experimental procedure

When state machines are used, participants are expected to generate test sets based on the round-trip path (RTP) testing technique [16], a common state testing strategy that can scale up to large state machines but that is more demanding than simply covering all transitions. A state machine would be represented as a tree graph called transition tree which includes (in a piecewise manner) all the transition sequences (paths) that begin and end with the same state, as well as simple paths (i.e., sequences of transitions that contain only one iteration for any loop present in the state machine) from the initial state to the final state. A procedure based on a breadth-first traversal of the state machine is used for deriving the transition tree. More precisely, during the traversal of the graph corresponding to the state machine, a tree node is considered terminal when the state it represents is already present anywhere in the tree or is the final state. The round-trip path testing technique corresponds to covering all paths from the start node to the leaf nodes in a transition tree. Since different transition trees can be generated from a state machine, we wanted to avoid having our results affected by variations due to alternative transition trees, or trees possibly wrongly constructed by participants. In practice, such trees would be generated automatically from state machines using a dedicated tool. For these reasons, one tree per cluster was provided to all participants working with the state machine model. Also, state machine

transitions with disjunct conditions were separated beforehand (one transition for each disjunct) to ensure full coverage of guard conditions. Transition trees are in theory supposed to be "equivalent" in the sense that they all cover (in a piecewise manner) the round trip paths.

Participants working with state machines were asked to manually generate test cases covering round-trip paths in the provided transition tree. They were also asked to use state invariants in their oracle assertions. After executing a transition, an oracle assertion checks the new cluster state with the expected state invariant. In replicated experiments, in addition to state invariants, participants were asked to implement contract assertions in their oracles. Contract assertions include class invariant, and methods' preconditions and postconditions. The different invariants and contracts were provided in OCL [4] in the documentation provided to participants.

For structural testing, participants were told to attempt covering all blocks (nodes, statements) and edges in the methods' control flow graphs of the original (not mutated) source code. This is a common practice when testing classes and it is therefore a realistic baseline of comparison for state testing. By running their drivers on the instrumented code, the participants identify non-covered nodes and edges. This guides participants to identify new test cases to be added to their drivers to improve structural coverage. Participants using structural testing were advised to write oracles checking expected output/attribute values against actual ones.

For each treatment, the test artifacts being used are:

a) *Code*, complemented with some textual comments to define the meaning of the most complex variables and methods. We also provided a high-level textual description of the cluster objectives and functionalities.

b) *State machine* describing the behavior of classes, plus the related public interface(s), class diagram, contracts and state invariants in OCL [4], and a high-level textual description of the software objectives and functionalities.

For both treatments, we were aware of the fact that coverage was unlikely to be complete as time was fixed and limited, and there was a wide variation in participant skills. However, we considered this is not avoidable in the context of a controlled experiment and decided, instead of addressing this threat, to account for it in the data analysis by using coverage (state machine and code) as an interaction factor. For source code, both node and edge coverage were used in the analysis. It is often the case that, when performing controlled experiments, one can assess a treatment effectiveness or its impact on the time needed to perform the task; it is often difficult to assess both at the same time [11]. In this work we focus on the effectiveness of testing strategies rather than on their impact on the time/effort needed by testers to develop test drivers.

Possible learning effects were simply measured during the data analysis by accounting for the laboratory in which the work took place. For each tested cluster, results from the first laboratory were compared to those of subsequent laboratories. Also, for each participant, effectiveness was compared between the labs. Participant ability was measured by considering the block to which they belonged as described in Section 3.2.1. The experiment only involved three class clusters and it is therefore not possible to analyze the impact of code and state machine characteristics on fault detection through statistical analysis. We, however, perform an in-depth, systematic qualitative analysis of why certain faults fail to be detected by test drivers in each cluster.

3.2.6 Experiment design

To avoid learning and fatigue effects or the specific class clusters to have a confounding effect with our experimental treatment, each participant group performed the experiment in two separate labs with a different class cluster under test and a different treatment. Table 3 shows the distribution of treatments among groups of participants. Each treatment is executed by two different groups of participants, in the first or the second laboratory (lab order). As a result, each group executed different combinations of treatment and class cluster in each lab. Such an experimental design is standard and is referred to as a balanced 2*2 factorial design [93].

	Group 1	**Group 2**	**Group 3**	**Group 4**
Lab 1	Cruise Control + State machine	OrdSet + State machine	Cruise Control + Code	OrdSet + Code
Lab 2	OrdSet + Code	Cruise Control + Code	OrdSet + State machine	Cruise Control + State machine

Table 3: Distribution of experiment treatments among groups

Test drivers submitted by the participants were executed offline on a set of mutant programs (Section 3.2.4) to measure their mutation scores, as further discussed in the next section. Test drivers were also executed on an instrumented version of the original code of the software under test to collect node and edge coverage data. The development and data collection were done on the Eclipse 3.0 platform [1]. Two Eclipse plug-ins, the Eclipse Test and Performance Tools Platform project (TPTP) [3] and the Eclipse Metrics plug-in [2], are used to collect cost-related data.

From a general standpoint, the notion of cost in the context of testing is complex and can be related to many factors such as test suite size, test case identification complexity, CPU user time usage, and time-to-market. In our context, we focus on generation and execution cost, and in

particular we rely on the number of method calls measured by executing instrumented test drivers, on the CPU user time needed by the test driver execution, and on the size of a test drivers measured in Lines of Code (LOC). Although these are clearly surrogate measures, number of calls [8, 14, 25, 53, 91], and CPU time [20] have been used in a number of testing studies. In these studies, one test case often corresponds to one execution of a function/program (e.g., [53]). This corresponds in our study to one execution of a method in the class cluster under test.

3.3 Experiment operation

3.3.1 Preparation and material

The students were first introduced to the class clusters under test during the experiment to make sure they solely relied on the documentation presented to them. To prepare the students for the different tasks required for the experiment—and thus make the experiment realistic in terms of human factors—the experiment was preceeded by a refresher course on the basics of testing (e.g., test cases, test sets, testing criteria, and test drivers), structural and functional testing, and class testing. Students applied the concepts and techniques they were taught in assignments and laboratory exercises prior to the start of the experiment's tasks.

To calculate node and edge coverage, the classes under test were instrumented using the Observer pattern [45] and by building the control flow graphs of their methods. The instrumentation code includes the definition of control flow nodes and edges, an Observer class that is informed of visited nodes and edges, and a Recorder class that generates coverage report.

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

The following documents were provided to all students in all groups: (1) Printed list of instructions to guide students through the different tasks to complete; (2) High-level description of the class cluster; (3) Eclipse tutorial; (4) Driver template (with slight variations depending on the testing strategy). Depending on the treatment (testing strategy), participants were provided with:

- *Source code (for participants using white-box testing).* To allow participants for easily computing node and edge coverage, we provided them an instrumented version of the code—packaged in a jar file—along with the original, non-instrumented, source code.
- *State machine diagrams (for participants using state-based testing)*, plus (a) class public interfaces; (b) a transition tree; (c) class diagrams, operations' contracts and state invariants in OCL; and (d) an executable jar file of the class cluster containing byte code only, and not the source code as we want to ensure pure model-based testing.

3.3.2 Execution

In an experimental, artificial setting, the time allocated to an experiment is necessarily limited. Test techniques are compared assuming a limited, equal effort on the part of testers. Although this does not allow the assessment of the technique's maximum fault detection effectiveness, it is necessary to perform a fair comparison among participants, and to investigate on the effectiveness of the techniques when applied under time constraints [19], a realistic scenario in industrial development environments. In our experiments each lab lasted three hours for two experiments for Carleton's experiments and four hours for Sannio's experiments.

Right before the lab, students were introduced to the class clusters under test during the experiment to make sure they solely relied on the documentation presented to them. Also, we explained the tasks to be performed during the experiment.

During each lab, students were first asked to read the documentation of the class cluster to understand the functionality it provides; then they were asked to write driver code following precise instructions, by identifying test cases based either on the provided transition tree (covering round-trip paths) or based on structural coverage criteria, depending on the group to which they belonged. In the latter case, students were asked to write method sequences capturing realistic scenarios in their test cases, and they were advised to use Equivalence class testing and boundary analysis [54] to help the identification of method parameter values. For structural testing, students were instructed to run their drivers on the instrumented version of the code to identify node and edge coverage; the report generated by the instrumented code identifies the non-covered nodes and edges, which can guide students to identify new test cases to be added to their drivers to improve structural coverage.

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

After completing their tasks, the participants were asked to answer and return a survey questionnaire. The questionnaire addresses three areas: the tasks implemented, the testing technique used and the work environment (Appendix L).

3.3.3 Data Collection

After the experiment, the test drivers produced by participants were executed on the original code of the two class clusters to inspect their correctness and to eliminate inadequate drivers which could not be used for experimental analysis, e.g., state machine drivers that did not implement any RTPs or structural drivers that did not have oracle implementations. The latter was necessary as it was impossible for us to ensure a satisfactory compliance of the participants with the lab task instructions. Selected drivers had to comply to the test technique implemented and include oracle assertions to detect faults.

Perl scripts were used to automatically execute drivers on mutants and on code instrumented versions, in order to collect data required to compute mutation scores, node and edge coverage, and distributions of undetected fault types. The Eclipse Test & Performance Tools Platform Project (TPTP) [3] plug-in was used to measure the number of method calls when executing test drivers. The Eclipse Metrics plugin [2] was used for statically counting the number of statements in collected drivers.

Survey data were collected manually into spreadsheets and analyzed in JMP to support our quantitative results.

3.4 Experiment replication

There are many reasons why replications are necessary in experimentation, and in particular for software engineering experiments [85]. In our context, replications were useful (i) to address or mitigate threats to validity discovered in the first experiment; (ii) to use an additional class cluster thus strengthening the external validity of our results; and (iii) to collect results from at least two distinct geographical locations and organizational settings, once again improving

external validity. Though the experimental design (shown in Table 3) did not change across replications, we did some context changes, summarized in Table 4, and motivated below.

	Experiment			
Design or context attribute	**Carleton 1**	**Sannio 1**	**Carleton 2**	**Sannio 2**
Cluster under test in addition to Cruise Control	OrdSet	OrdSet	Elevator	Elevator
Participants level	Undergrad	Graduate	Undergrad	Graduate
State invariants in oracles (state testing)	Yes	Yes	Yes	Yes
Contract assertions in oracles (state testing)	No	Yes	Yes	Yes
Lab time length	3 hrs	4 hrs	3 hrs	4 hrs
Common documentation (for both techniques)	During lab	Before lab	During lab	Before lab
Use case diagrams added to documentation	No	Yes	Yes	Yes
Including oracle helper class to template driver	No	Yes	Yes	Yes
Implicit Sneak path testing (Cruise Control)	No	Yes	Yes	Yes

Table 4: List of design and context changes across experiments in the series

Using different objects. In both the Carleton 2 and Sannio 2 experiments we replaced `OrdSet` with `Elevator`, which is a control class cluster (eight classes) for an elevator control system. This was motivated by the first two experiments, Carleton 1 and Sannio 1, which showed clear limitations for state testing when applied to real-time control classes (`Cruise Control`). Therefore, we wanted to further explore the matter on another, more complex instance of real-time class cluster to make sure the limitations we had observed were not specific to `Cruise Control` and also investigate how increased complexity would affect the results. Both `Elevator`'s model and code are more complex than those of `Cruise Control`.

Extending the RTP criterion to cover sneak paths. Our qualitative analysis of undetected faults in Carleton 1 (Chapter 5) pointed to a deficiency in the state testing technique. In its original form, as it is common practice, `Cruise Control`'s state machine did not include implicit transitions (self-transitions with no actions, named sneak-path in [15]), that are not taken

into consideration by the RTP criterion As a result, many faults remained undetected though exercizing sneak paths would have clearly uncovered them. Therefore, in all three replications, the testing of sneak paths was made part of our state test strategy in addition to round-trip paths.

Use of different oracles. In the last three experiments (Sannio 1, Carleton 2 and Sannio 2), participants working with state testing were instructed to use contract assertion in addition to state invariant assertions in their oracles. That is because a qualitative analysis of undetected faults (Chapter 5) from Carleton 1's results suggested that the use of contract assertions in oracles would have led to the detection of a significant number of these faults (the average mutation score of `OrdSet` drivers went from 50% in Carleton 1 to 72% in Sannio 1).

Providing support to write oracles. In Carleton 1 we noticed that the implementation of state invariants (from OCL to Java code) took considerable lab time. Therefore, to give participants more time to implement their drivers, we provided them with an oracle class helper implementation in addition to the provided template driver in each replication. For each system, the oracle helper class included methods for checking the class invariant, state invariants, preconditionss and postconditions for methods. Participants working with structural testing were provided with an implementation of an oracle method comparing all attributes values against expected ones. It could be simply used in its original form or simplified, by comparing only a subset of attributes, to implement more specific oracles. For instance, when calling a public method that is expected to change one attribute only, the tester may prefer to implement an oracle checking method that verifies the resulting value of that attribute without calling the oracle checking method that verifies all attributes. This would lower the number of called methods.

Using longer labs. Since survey questionnaires from Carleton 1 experiment (Appendix M) suggested a lack of time to perform the experimental tasks, whenever possible, i.e., for Sannio 1

and 2 experiments, we used laboratories of four hours instead of three. In addition, class clusters were introduced in the morning, before the four laboratory hours started, so that participants had 4 hours to be entirely dedicated to the experimental tasks. On the other hand, we are aware that a longer laboratory could increase the risk of fatigue effects.

The number of participants in each experiment was constrained by the number of registered students in the corresponding testing course in which the experiment took place. Furthermore, only a subset of the collected drivers could be accounted for in the analysis. Some drivers did not comply with the instructions we provided and it would have therefore blurred the results of the experiment if they had been taken into account. For instance, there were state drivers that did not implement RTPs and did not have oracle implementations. Table 5 provides a summary of the number of participants and the total number of valid drivers.

	Total # of participants	**Number of participants in:**				**# valid drivers for OrdSet**	**# valid drivers for Cruise Control**	**# valid drivers for Elevator**
		Group 1	**Group 2**	**Group 3**	**Group 4**			
Carleton 1	48	11	13	12	12	34	32	NA
Sannio 1	25	8	7	6	4	24	24	NA
Carleton 2	19	4	5	5	5	NA	16	11
Sannio 2	19	5	5	4	5	NA	18	19

Table 5: Participants and drivers distribution across experiments

In summary, given the replication classification provided in [11], our replications fall into the following categories: (1) replications that do not vary any research hypothesis but that vary the experiment procedure and material (e.g., changing lab duration and class clusters) and (2) replications that vary research hypotheses by varying variables intrinsic to the object of study (e.g., contract assertions in oracles and sneak path testing).

Table 6 below summarizes the number of valid (structural and state) drivers, and the number of mutants we use for each experiment. The number of mutants per cluster can be regarded as an indicator of the complexity of its code.

Experiment	Cluster	Number of mutants	Number of observations	
			structural	state
Carleton 1	OrdSet	624	17	17
	Cruise Control	382	15	17
Sannio 1	OrdSet	624	11	13
	Cruise Control	382	12	12
Carleton 2	Elevator	1176	4	7
	Cruise Control	382	10	6
Sannio 2	Elevator	1176	10	9
	Cruise Control	382	8	10

Table 6: Summary of number of observations and mutants across experiments and clusters

3.5 Overview of statistical analysis

We provide here a short overview of the statistical techniques we applied to address research questions.

Univariate analysis was performed to assess the isolated effect of an independent variable on a dependent variable. In particular, two-sample *t*-tests were performed to compare test techniques in terms of fault-detection effectiveness and cost, and determine whether differences in means could be due to chance. The level of significance is set to $\alpha = 0.05$ for all tests, though we also report p-values.

To avoid potential threats due to the violation of the *t*-test assumptions, equivalent non-parametric tests (Wilcoxon rank sum tests [37]) were also performed to verify that negative results are not due to strong departures from the normality assumptions, which are known to make *t*-tests conservative. In the rare cases where different results are observed, this is clearly

stated. The t-test is a parametric test and requires data to be normally distributed. We use the Anderson-Darling test to check for normality [41] (H_0: samples significantly deviate from a normal distribution) to verify the normality of the data studied. We report normality results whenever samples deviate significantly from the normal distribution. Fortunately, in experiments where a significant deviation from normality was found, the number of participants (and thus data points) was large enough (48 and 25) to make parametric tests such as t-test and ANOVA robust to the encountered normality deviation [86].

Performing multiple tests on the collected sample data to address a number of distinct hypotheses may introduce chance capitalization of type I error. A number of statistical corrections were proposed in literature to address this issue. Among these is the Bonferroni correction which states that if an experimenter is testing n dependent or independent hypotheses on a set of data, then the statistical significance level that should be used for each hypothesis separately is $1/n$ times what it would be if only one hypothesis was tested [47]. The Bonferroni correction is considered to be very conservative. While addressing type I error capitalization, the correction has a serious drawback: an increased likelihood of type II error, that no effect or difference is declared while in fact there is an effect. In addition, the correction ignores the correlation between the considered tests [48]. The Bonferroni correction is strongly criticized by a number of researchers [38, 47, 79, 89]. The rough false discovery rate [66] is another statistical method used in multiple hypotheses testing to correct for multiple comparisons with a less restrictive criterion than the Bonferroni adjustment. The adjusted alpha for n distinct tests is calculated as: alpha * (n + 1) / 2n.

In our context, we apply the Holm's procedure [48] to the data pooled from all four experiments. It is suitable to small sample sizes (as in our case), is less restrictive than Bonferroni, and accounts for the correlation between tests. Specifically, it sorts p-values, in ascending order,

resulting from the n distinct tests, and multiplies the smallest p-value by n, the next by n-1, and so on. Finally, to be compared to the significance level, given the resulting ordering index i, p-values are corrected as follows: $p_i = max_{j \leq i}(p_j)$.

Not all hypotheses addressed in this research are considered for the adjustment. *RQ3*, *RQ5*, *RQ6* and *RQ9* are not analyzed with t-tests, and are therefore not considered. *RQ4* is considered twice as it addresses the complementariness of both state testing and structural testing in terms of fault detection. *RQ7* and *RQ8* are considered three times each for each considered cost measure. Therefore the total number of distinct tests considered for the correction is $n = 10$. As the samples in each of our experiments are small, we only apply the Holm's procedure to the data pooled from all four experiments.

When dealing with small sample sizes, it is also interesting to look at the *power* of statistical tests. The power of a statistical test is the probability that the test will reject a false null hypothesis (that it will not make a Type II error) and is a function of three factors: the specific statistical test used (e.g., *t*-test), sample sizes, sample standard deviations for the different treatments, and the selected effect size[7] [71]. There are several potential applications for power analysis [40], among others to determine an appropriate sample size for an experiment. However, in our case this is not possible (the sample size is dictated by the availability of participants). Instead, we used the power analysis for a different purpose, i.e., to determine the minimum effect size above which we achieve a certain power, typically 80%, and thus above which we can be confident about our conclusions. In other words, if we do not observe any statistically significant result, for small sample sizes, we cannot claim there is no effect of the treatment. We can, on the

[7] Effect size is a measure of the strength of the relationship between two variables. It is often useful to know not only whether an experiment has a statistically significant effect, but also the size of any observed effects. Cohen's *d* [33] is an appropriate effect size measure to use in the context of a t-test on means, where *d* is defined as the difference between two means divided by the pooled standard deviation for those means.

other hand, be confident that there is no effect above a certain effect size for which the power reaches 80% or more.

To visualize distributions and allow for comparing results, we use both mean diamonds and box plots. A mean diamond indicates the sample's mean and 95% confidence interval and whether this is significantly different from other samples. Box plots show selected quantiles of continuous distributions and extreme values. In particular, boxes span between the 25^{th} and 75^{th} percentiles, also called *quartiles*. The line across the middle of the box identifies the median sample value. The dashed lines, sometimes called *whiskers*, are placed at a distance equal to 1.5 times the interquartile distance below the 25^{th} percentile and above the 75^{th} percentile respectively. A box plot may also show a mean diamond.

The analysis of co-factors effects, and their interaction with the treatments, is done using a two-way analysis of variance (ANOVA) and a bivariate least-squares regression [37]. The aim is to study the simultaneous effect of the test technique on fault detection and its interactions with other factors (e.g., coverage). This is important as the effect of test techniques can vary widely based on factors related to class cluster and state machine characteristics, participant ability, and so on.

Chapter 4

EXPERIMENTAL RESULTS

This chapter presents the results obtained from the four controlled experiments. Results of each experiment are presented separately, however identifying and discussing commonalities and differences. Wherever necessary, the analysis of combined datasets of all experiments is also performed to increase the statistical power of the performed tests.

In the first section, we compare drivers' mutation scores for the two testing strategies under investigation, thus addressing RQ1. We also compare the coverage level of drivers generated based on state machines to those based on control flow coverage in source code (RQ2). Then, we analyze interaction effects of the test techniques with code coverage and cluster characteristics (RQ3). In Section 4.2, we focus on the complementariness of test techniques in terms of fault detection effectiveness, and the benefits of combining them (RQ4 and RQ5). We further refine this analysis by looking individually at each mutation operator (RQ6). Cost and cost-effectiveness analyses are reported in Section 4.3 (RQ7 and RQ8). The impact of equivalent mutants on the results of previous sections is reported in Section 4.5. Finally we analyze the collected survey data in Section 4.6.

4.1 Impact of test techniques on fault detection effectiveness

This section discusses the impact of the independent variable "test technique" (structural vs. state machine) on the dependent variables "fault-detection effectiveness" and "code coverage" (RQ1, addressed in Section 4.1.1). Next, we investigate the possible interactions between the test technique and a number of factors in terms of their impact on fault detection effectiveness (RQ2, addressed in Section 4.1.2). These interaction factors include: code coverage, lab order, participant ability, and cluster characteristics.

4.1.1 Univariate analysis

To address RQ1, we perfomed a univariate analysis of the fault detection ratios of test drivers (Section 4.1.1.1), and a discussion of the impact of cluster properties on the fault detection effectiveness of each test technique (Section 4.1.1.2), then we discuss mutation score variation across drivers' (Section 4.1.1.3) and present a univariate analysis of drivers' code coverage (Section 4.1.1.4) across test techniques. Code coverage analysis complements fault detection analysis by providing a possible explanation for differences in fault detection.

4.1.1.1 Fault detection analysis

a. Comparison of mutation scores means across test techniques, clusters and experiments

Boxplots showing mutation scores' distribution for each cluster, experiment, and treatment are presented in Figure 2, Figure 3, and Figure 4. More details on descriptive statistics of the drivers' mutation scores can be found in Table 52, Appendix E. Descriptive statistics of drivers' mutation scores show that structural drivers for the three tested clusters performed better at fault detection than state drivers except for `OrdSet` in Sannio 1 where the state machine mean mutation score was slightly higher than for structural drivers (72% vs. 70%). We notice from the results the high

variability of mutation scores achieved by structural drivers. The trend is different in state drivers where little variability is observed in `Cruise Control`'s drivers, though high variability is observed in `OrdSet` and `Elevator`. This suggests that the complexity of the code of the cluster under test and the complexity of its state machine have an impact on mutation score, as discussed in Section 4.1.2.

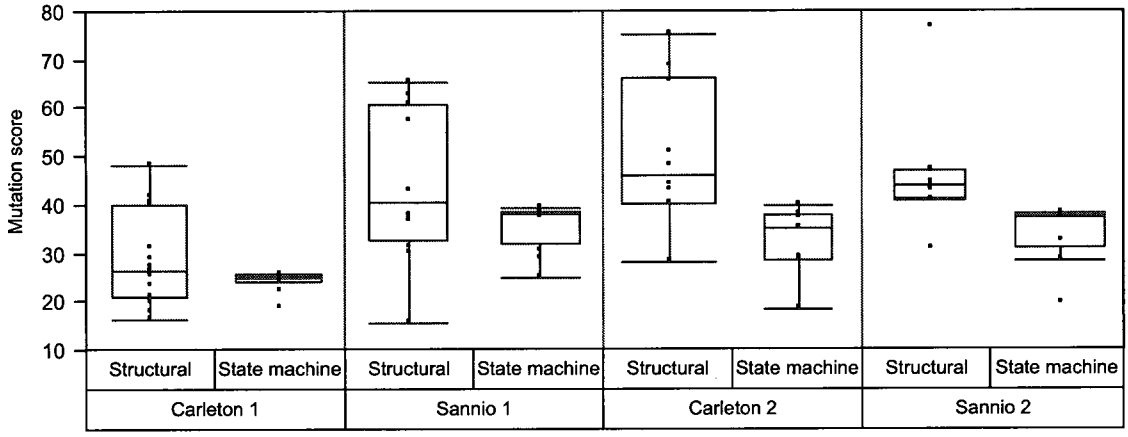


Figure 2: Cruise Control's drivers' mutation scores distribution

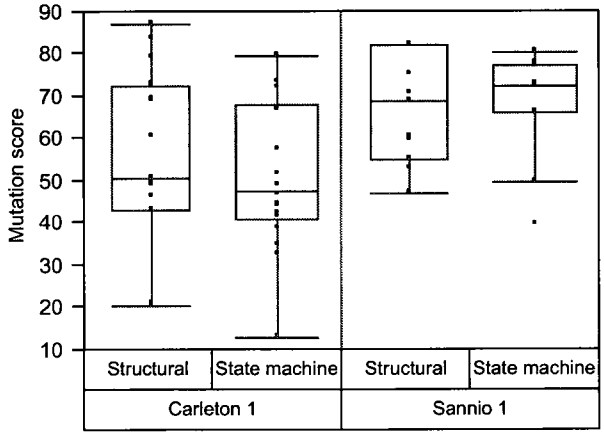


Figure 3: OrdSet's drivers' mutation scores distribution

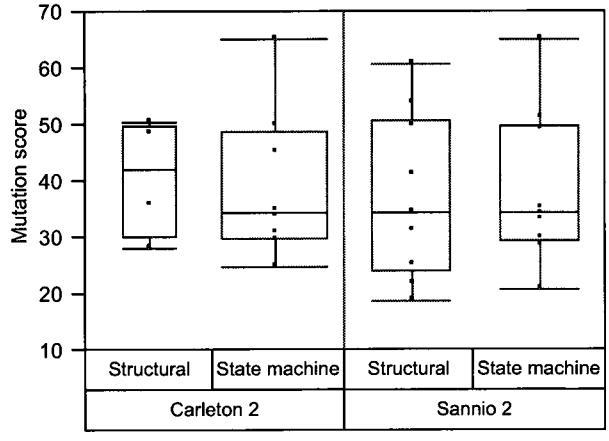


Figure 4: Elevator's drivers' mutation scores distribution

Table 7 reports on the results of a two-tailed *t*-test [37] for each class cluster to assess the statistical significance of the difference in terms of mutation scores between the two test techniques. The following null hypothesis is tested: "*There is no significant difference between*

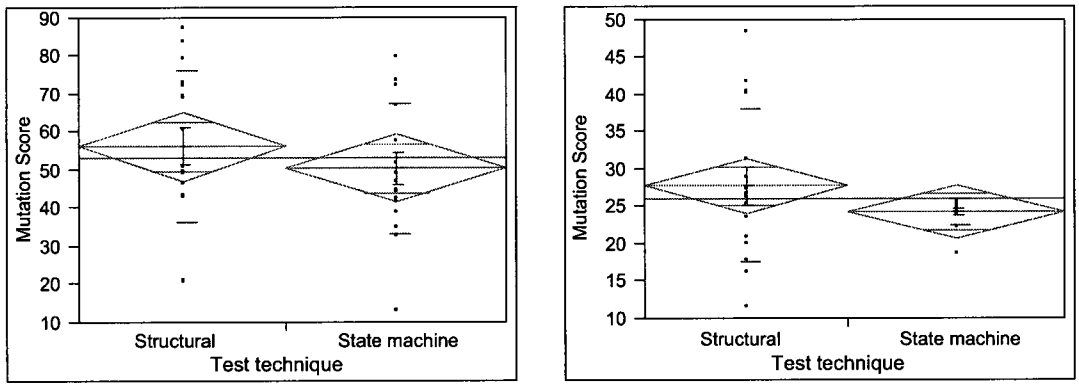
the number of faults detected by state test cases (Ts) and structural test cases (Tc)". The results are reported in Table 7 where each row corresponds to statistics for one cluster in an experiment. Columns show the degree of freedom[8] (DF), the mutation scores' mean and variance per treatment, the calculated t-value, and the corresponding p-value (Pr > |t|). The Anderson-Darling test [41] (H_0: samples significantly deviate from a normal distribution) indicated that mutation scores are normally distributed for the Carleton 2 experiment (p-value=0.07) and Sannio 2 (p-value=0.19), while they significantly deviate from normality in Sannio 1 (p-value=1e-6) and Carleton 1 (p-value=0.002). In addition to t-tests, we therefore performed a non-parametric Wilcoxon test to ensure the validity of the t-test results in cases where normality assumptions are violated. Consistent results were obtained with those reported in Table 7.

Experiment	Cluster	DF	Mutation score mean		Mutation score variance		t-value	Pr > |t|
			Structural	State	Structural	State		
Carleton 1	OrdSet	31.6	56.15	50.27	399.42	295.97	0.93	0.359
	Cruise Control	15.8	27.69	24.47	92.07	2.86	1.35	0.197
Sannio 1	OrdSet	20.73	70.31	71.96	161.1	154.1	0.314	0.7567
	Cruise Control	12.97	44.65	35.65	257.5	23.3	-1.86	0.0853
Carleton 2	Elevator	5.46	40.54	35.44	110.9	78.15	-0.82	0.4483
	Cruise Control	13.77	50.13	30.8	219.4	55.17	-3.46	**0.0039**
Sannio 2	Elevator	15.92	36.97	35.06	198.9	172.7	-0.35	0.7325
	Cruise Control	9.37	46.89	34.55	173.4	36.79	-2.45	**0.0358**
All experiments	OrdSet	55.75	61.71	58.94	345.96	348.72	-0.56	0.5742
	Cruise Control	60.17	40.97	30.82	245.97	47.72	-4.04	**0.0001**
	Elevator	26.18	37.99	38.55	166.08	128.61	0.12	0.9033

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

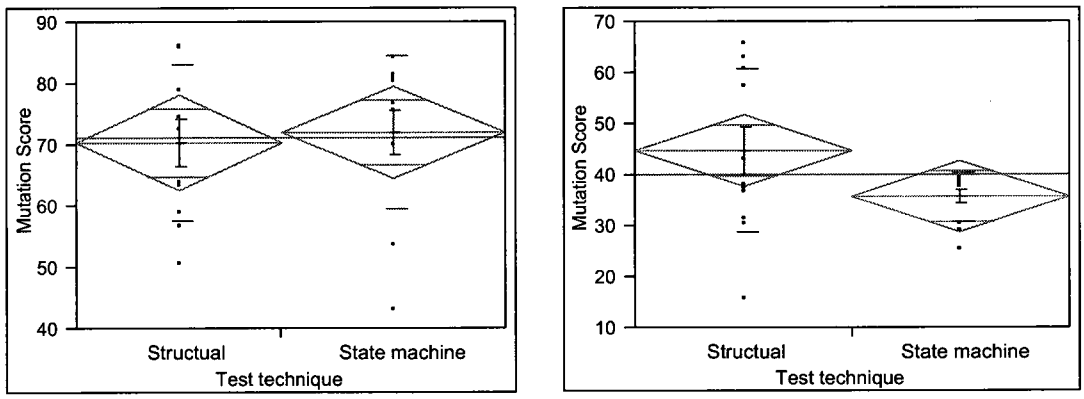
[8] The number of degrees of freedom is the number of independent observations for a dataset. DF is explained as the number of independent components minus the number of parameters estimated [88]. There are various ways in which the degrees of freedom for the two-sample *t*-test can be calculated. In the case of un-equal variances Welch's *t*-test is mostly used. The degrees of freedom for this method are calculated with a complex formula. Note that the degrees of freedom in a two-sample test are a weighted average of the degrees of freedom in the separate samples.

Figure 5, Figure 6, Figure 7 and Figure 8 show mean diamonds of the mutation scores of test drivers. This visual representation helps compare the means of mutation scores of state drivers to those of structural drivers. A means diamond depicts the sample mean and 95% confidence interval. The line across each diamond represents the group mean. The vertical span of each diamond represents the 95% confidence interval for each group.



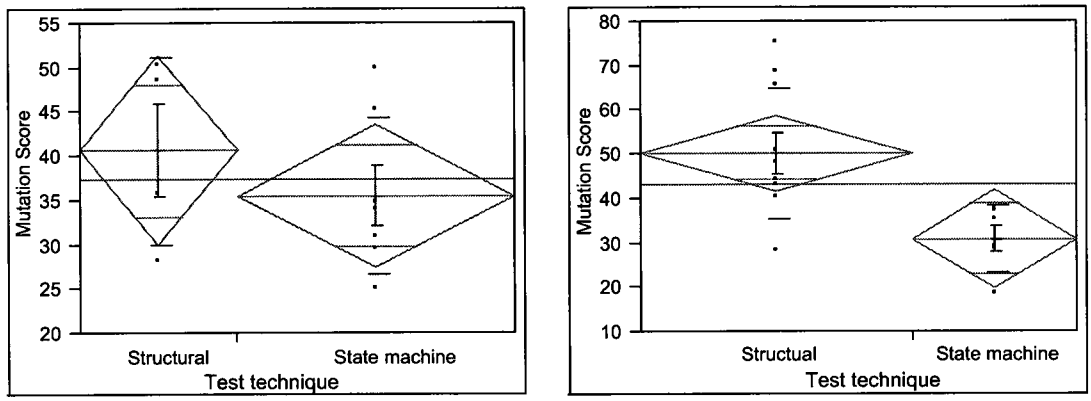
a) OrdSet b) Cruise Control

Figure 5: Distributions of Carleton 1 mutation scores



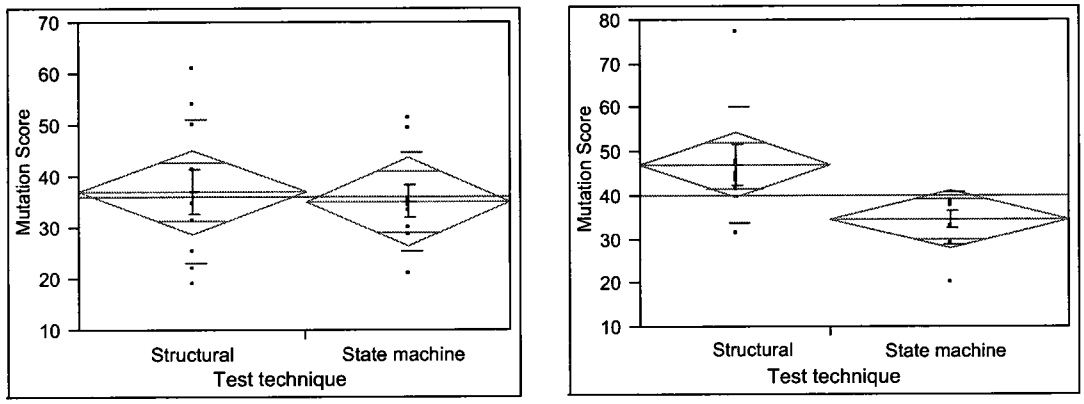
a) OrdSet b) Cruise Control

Figure 6: Distributions of Sannio 1 mutation scores



a) Elevator b) Cruise Control

Figure 7: Distributions of Carleton 2 mutation scores



a) Elevator b) Cruise Control

Figure 8: Distributions of Sannio 2 mutation scores

For both `OrdSet` and `Elevator` a *t*-test yielded a *p*-value greater than our selected level of significance and therefore the null hypothesis cannot be rejected. No statistically significant difference between state and structural drivers can be observed for these two clusters in terms of mutation scores. `Cruise Control` structural drivers had significantly higher mutation score means than state drivers in the last two experiments. To further investigate this difference between the results of the first two and the last two experiments, recall the differences between replications: After the first experiment, recall that state testing was improved by performing sneak-path testing and oracle precision was increased to include contract checking. The state

mutation score did not vary significantly between Sannio 1 and Sannio 2 where all design factors were the same (same lab time, oracle precision, and participants' skills). The mutation score mean slightly dropped in Carleton 2, from that observed in Sannio 1 and Sannio 2, and this may be attributed to the shorter lab time (three hours instead of four). However, the variance in mutation scores of state drivers varied significantly between Sannio 1 on one hand and Carleton 2 and Sannio 2 on the other hand. This can be attributed to the varying number of implemented RTPs in Carleton 2 (few participants implemented all RTPs – Table 53) and to the varying levels of implemented oracles in Sannio 2 (only state invariants, state invariants + class invariants, or state invariants + contracts checking). In Sannio 1, almost all participants implemented all RTPs from the `Cruise Control` transition tree and consistently used state and class invariants as oracles without resorting, in most cases, to pre- and post-conditions. On the other hand, structural mutation scores in the last two experiments were higher than in Sannio 1, especially in Carleton 2, with less variance.

These results were confirmed also when applying the Holm's correction (Section 3.5) procedure to the pooled data from all experiments to account for possible type I error chance capitalization.

b. Accounting for trivial mutants in the calculated mutation scores means

To further investigate the fault detection effectiveness of each technique, we computed again mutation scores after removing "trivial mutants" i.e., mutants that are far too easy to be detected, and thus not to be considered realistic and representative of real faults. In particular, we classified as trivial, mutants killed by all drivers. The number of trivial mutants was 2 (0.3%) for `OrdSet`, 12 (3.1%) for `Cruise Control`, and due to its significantly larger size, was much higher (144 or 12.2%) for `Elevator`. We re-computed t-tests considering mutation scores for non-trivial

mutants only. The results presented in Table 8 are consistent with those obtained when including trivial mutants. This further supports the validity of our results.

Experiment	Cluster	DF	Mutation score mean		Mutation score variance		t-value	Pr > |t|
			Structural	State	Structural	State		
Carleton 1	OrdSet	31.64	56.01	50.11	401.99	297.88	-0.93	0.3593
	Cruise Control	14.77	28.55	23.96	92.66	2.88	-1.82	0.0887
Sannio 1	OrdSet	20.73	70.21	71.86	162.14	155.09	0.314	0.7567
	Cruise Control	12.97	43.61	34.6	257.47	23.3	-1.86	0.0853
Carleton 2	Elevator	5.46	32.24	26.44	144.04	101.48	-0.81	0.4483
	Cruise Control	13.77	48.51	28.56	233.88	58.81	-3.46	**0.0039**
Sannio 2	Elevator	17.91	28.18	29.43	258.26	224.25	0.18	0.8592
	Cruise Control	9.37	45.17	32.43	184.81	39.22	-2.45	**0.0358**
All experiments	OrdSet	55.75	61.58	58.81	348.2	350.97	-0.56	0.5742
	Cruise Control	60.17	39.95	29.47	240.69	43.12	-4.17	**<.0001**
	Elevator	26.18	29.33	28.19	215.67	166.49	0.12	0.9034

Table 8: *t*-test results for the mutation score comparison after removing trivial mutants

c. Power analysis

In some cases, the lack of significant results in Table 7 may be due to a lack of statistical power due to small samples. The probability that we committed a Type II error by accepting H_0 and rejecting H_1 needs to be evaluated by conducting a statistical power analysis. In our context, since we do not know and cannot estimate the actual effect size, power analysis is used to assess what would have been the minimum effect size for which we have a reasonable chance of detecting a statistically significant result. Table 9 presents the power analysis results.

Experiment	Cluster	Sample size		Mean difference	Standard deviation		Observed effect size	Effect size at 80% power
		Struct.	Stat.		Struct.	Stat.		
Carleton 1	OrdSet	17	18	5.88	19.99	17.2	0.32	0.98
	Cruise Control	15	17	3.22	9.59	1.69	0.47	1.03
Sannio 1	OrdSet	11	12	-1.65	12.69	12.41	0.13	1.23
	Cruise Control	12	12	9	16.05	4.83	0.76	1.20
Carleton 2	Elevator	4	8	5.1	10.53	13.31	0.42	1.90
	Cruise Control	10	7	19.33	14.81	7.54	1.64	1.48
Sannio 2	Elevator	10	10	1.91	14.1	13.14	0.14	1.32
	Cruise Control	8	11	12.34	13.17	5.95	1.21	1.38
All experiments	OrdSet	28	30	2.77	18.6	18.67	0.15	0.74
	Cruise Control	45	47	10.15	15.68	6.95	0.84	0.59
	Elevator	14	18	-0.56	12.89	12.83	0.04	1.03

Table 9: Estimated effect size for 80% power

We calculated the minimum effect size required to achieve a power of 80% at a significance level of 0.05. We also calculated the observed effect size based on the calculated mean difference and the pooled standard deviation.

We notice that for all four experiments the minimum effect sizes corresponding to 80% power are large and this further justifies our decision to pool together the data of all experiments, which shows to significantly increase power (i.e., decrease the minimum effect size matching 80% power). In fact, the cases where results are significant (`Cruise Control` for Carleton 2, Sannio 2, and all experiments) correspond to the cases where the actual effect size is close or above the 80% power effect size threshold.

We can therefore conclude that this experiment does not allow us to draw conclusions about modest effect sizes. If there are significant differences, other than those already reported, between structural and state testing mutation scores, the unknown, actual effect size is

significantly lower than 1, e.g., lower than 0.59, 0.74, or 1.03 for all experiments combined (last three rows and last column in Table 9).

4.1.1.2 Impact of cluster properties on mutation scores

Here we investigate whether the properties of the three class clusters used in the experiments had an impact on the fault detection effectiveness of the applied testing techniques.

For `OrdSet`, the state machine provides an accurate description of the behavior of the class and despite the complexity of its state machine, the functionality of `OrdSet` is rather intuitive for computer science and software engineering students. Understanding `OrdSet`'s state machine and code was far easier than for `Cruise Control` or `Elevator`, thus resulting in higher mutation scores.

For `Cruise Control`, we observe a low mutation score and low code coverage for both state machine and structural drivers. For the former, one plausible reason is that its state machine does not provide a description of how values of class attributes are calculated and updated over time. The `Cruise Control` state machine models the cluster's functionality without modeling its real-time properties: i.e., it does not provide a description of how values of class attributes (e.g., speed and distance) are calculated and updated over time. This was a choice of the original designers to avoid an overly complex and difficult to understand state machine that would attempt to model the complex algorithms managing the variation of class attribute values over time with transitions triggered by timers (as exemplified in Figure 34, Appendix A).

We believe that such a highly complex state machine would be an unlikely modeling exercise to occur in practice. From a practical standpoint, modeling a state machine would be limited to state-dependent behavior: Implicit transitions that do not affect current state and bear no

actions—as for timer-controlled transitions of `Cruise Control`—are usually left out from the state machine to avoid cluttering the state machine diagram [16]. In the case of `Cruise Control`, timer controlled transitions do not affect the current abstract state of the cluster as defined in the state machine (Figure 33). In addition, the different actions associated with these transitions are of high computational complexity. Therefore, they were omitted from the state machine. A more practical and elegant solution to model the real-time properties of this cluster while avoiding excessive complexity of the state machine would be to model real-time properties in another diagram than the state machine. The aim would be to separate concerns, i.e., state-dependent behavior vs. time dependent behavior, to avoid designing complex artifacts while providing a model that represents closely the cluster. Additionally, some authors suggest that test models (e.g., state machine) that are built for testing purposes only do not have to specify all the behavior of the system under test. Test models are abstractions that contain only aspects that are deemed important for testing, and several such partial models are often preferred to one complete model [90].

A number of notations can be considered to separately model time dependent behavior. One possibility it to use an activity diagram that models the variation of cluster attributes as a function of time: see for instance Figure 29, Section 7.1 for the `Cruise Control`. An alternative could be timing diagrams to model state changes over time. In the case of `Cruise Control`, our conclusion is that a timing diagram is less adequate than an activity diagram because (1) the speed attribute is updated every 200 ms following a complex algorithm (which would be difficult to model in timing diagram), and (2) state changes are not time dependent.

Participants working with the `Cruise Control` code also had difficulties—confirmed in survey questionnaire answers Appendix M—to understand the system real-time properties without the

availability of a proper documentation or models. Basically, we did not provide any description of clusters' real-time properties to any of the groups. Based on answers to questionnaires collected after the experiment, 35% of the participants spent less than 25% of the laboratory time on understanding the cluster (model or code), around 40% of the participants spent up to 50% of the laboratory time on this same task, and only few participants had to spend most of the laboratory time on understanding the cluster (Table 96 in Appendix M). Alhough few participants noticed a relationship between the time factor and a number of class attributes, it was hard for them to understand the real-time algorithm that manages the class attributes solely based on code. A thorough understanding would have required complex reverse engineering or the availability of models such as communication diagrams, activity diagrams or timing diagrams thoroughly describing real-time properties.

The `Elevator` cluster also features a real-time behavior and includes concurrency. In contrast with `Cruise Control`, the real-time properties of `Elevator` not only affect class attribute values (concrete state), but also its current state as modeled by its state machine (abstract state). In `Cruise Control`, the speed and distance of a car in the running state vary as a function of time and current state. The state of the cluster remains the same over time though. However, in `Elevator`, the state of the `Elevator` may change as time elapses. For instance, a moving elevator may arrive to its destination and stops. In this case, the state of the elevator would change from moving, to finding next destination, and finally to the idle state. In other words, the real-time properties of `Elevator` are naturally modeled as state transitions.

State drivers' mutation scores in `Elevator` were relatively low (35.44 and 35.06 in Carleton 2 and Sannio 2), although the real-time behavior of `Elevator` was modeled in its state machine. This may be attributed to the following two reasons: (1) The `Elevator`'s state machine is

considerably more complex than the `Cruise Control`'s state machine. Even when lab time was increased in Sannio 2, results did not vary from those of Carleton 2; (2) Concurrency was not modeled by the state machine. Among the three tested clusters in this experiment, the concurrency aspect was specific to `Elevator`. Its state machine did not model concurrency between `Elevator` instances as it models only one instance. This would have an impact on the mutation scores of state drivers.

`Elevator`'s structural drivers did not do any better than state drivers (40.54 and 36.97 in Carleton 2 and Sannio 2). `Elevator`'s source code is far more complex than that of the other two clusters (Table 1) and therefore, understanding its concurrency and real time behavior solely from code is not easy.

Both laboratory time and oracle precision have an important impact on mutation scores. This is clear when comparing the results of Carleton 1 to the results of the subsequent replications, where mutation scores for both clusters and treatments increased significantly compared to those in Carleton 1. Recall that laboratory time was increased in Sannio 1 & 2 and oracle precision was increased in the three replications by including contract assertion checking.

We performed a two-tailed *t*-test, for each experiment, and for each treatment, to investigate the impact of the cluster under test on mutation scores (Table 10). We also performed t-tests on pooled data from each pair of experiments involving the same clusters.

Experiment	Test technique	DF	Observed effect size	Mutation score mean		Mutation score variance		t-value	Pr > |t|
				Cruise Control	OrdSet / Elevator	Cruise Control	OrdSet / Elevator		
Carleton 1	Structural	23.61	1.67	27.69	56.15	92.07	399.42	5.02	**<.0001**
	State machine	16.39	1.49	24.47	50.27	2.86	295.97	6.22	**<.0001**
Sannio 1	Structural	20.59	1.7	44.65	70.31	257.5	161.1	4.26	**0.0004**
	State machine	14.25	3.72	35.65	71.96	23.3	154.1	9.44	**<.0001**
Carleton 2	Structural	7.96	-0.65	50.13	40.54	219.4	110.9	-1.36	0.21
	State machine	11	0.52	30.8	35.44	55.17	78.15	1.03	0.3255
Sannio 2	Structural	15.55	-0.69	46.89	36.97	173.4	198.9	-1.54	0.144
	State machine	13.24	0.06	34.55	35.06	36.79	172.7	0.13	0.894
Carleton 1 & Sannio 1	Structural	51.34	1.53	35.83	61.71	223.17	345.97	5.69	**<.0001**
	State machine	36.37	2.14	28.93	58.94	43.71	348.72	8.28	**<.0001**
Carleton 2 & Sannio 2	Structural	28.93	0.8	48.69	37.99	190.3	166.08	-2.26	**0.0316**
	State machine	25.76	0.37	33.52	36.99	43.63	128.22	1.09	0.2857

Table 10: *t*-test results for mutation score comparison differences across clusters

Differences between `Cruise Control` and `OrdSet` are clear and significant, the latter showing much higher mutation scores. These results are further confirmed when pooling the data from Carleton 1 and Sannio 1. However, when comparing `Cruise Control` and `Elevator` in the last two experiments, we see no significant difference in terms of mutation scores between these two clusters, for both test techniques. When pooling data from Carleton 2 and Sannio 2, there is however a significant difference in mutation scores for structural drivers. This can be attributed to the difference in source code complexity of the two clusters. State drivers achieved low mutation scores for both `Cruise Control` and `Elevator`, though for different reasons as explained above.

4.1.1.3 Understanding mutation scores variation across collected drivers

Results highlight a high variability of mutation scores achieved by structural drivers. This is not always the case of state drivers, for which little variability is observed at least for `Cruise Control`'s drivers. The variation in drivers' mutation scores was measured as a standard deviation and had the lowest value for state drivers in `Cruise Control` / Carleton 1 (Table 52). When applying the RTP state based technique, participants had to produce test cases following a well defined criterion, and using a provided transition tree: a decision we made to ensure the conformance of test suites with a correct transition tree, which should in practice be automatically generated from the state machine (Section 3.2.3), this leaves little degree of freedom to the tester, also considering that – for Cruise Control – transitions have no guard conditions and require no parameter setting. Therefore, by following the RTP technique, similar results should be obtained by all participants resulting in a small standard deviation. The differences in mutation scores are due to variations in participants' ability, which lead to incorrect or incomplete implementations of state invariants in oracles, or the incomplete coverage of the transition tree. For instance, a number of participants simplified `Cruise Control` state invariants to only account for the state variable (`controlState`). Note that in addition to the `controlState` attribute assertion, the state invariant includes other attribute assertions such as assertions related to distance, speed and throttle values. The standard deviation of state drivers' mutation scores in the replications was higher than in Carleton 1 (between 5% and 8%) and this is most likely caused by the increased precision of oracles. Different participants, however, implemented different levels of oracle precisions, ranging from state invariants, state invariants + class invariants, to state invariants + contract checking.

As opposed to `Cruise Control`, the mutation scores' standard deviation for `OrdSet` state drivers is fairly large (17% as opposed to 1.5% for `Cruise Control` in Carleton 1). This can be explained as follows: (a) only few participants were able to cover all RTPs (35% RTP coverage on average) and test cases in their drivers covered various numbers of RTPs [70], (b) the state machine has complex guard conditions and requires parameter settings which introduce variation in test cases, (c) some faults can be detected only with very specific parameter values or set content, and (d) oracles often feature wrong or incomplete implementation of state invariants. The standard deviation for `OrdSet` state drivers decreased in Sannio 1 to 12%, which can be attributed to the increased lab time, allowing participants to cover more RTPs. We conjecture that if testers had unlimited time to complete the implementation of all RTPs as described in the test technique, the standard deviation would further decrease and only depend on points (c) and (d) listed above.

For `Elevator`, the state drivers' mutation scores variation was limited (around 9%). The high complexity of the state machine was the main source of variation as only few participants covered high numbers of RTPs. Following one transition tree, and implementing similar levels of oracle complexity (mainly state invariants + class invariants) limited the variation among drivers.

Overall, for all class clusters and experiments, variations in structural drivers' mutation scores were higher than for state drivers' mutation scores. While participants working with state machines had a precise strategy to follow, and pre-determined number of test cases (corresponding to the number of paths in the transition tree) to implement and pre-defined oracles, participants working with code were only required to achieve edge coverage, without further guidelines or instructions.

4.1.1.4 Code coverage analysis

a. Comparison of source code node and edge coverage across test techniques and experiments

Figure 9 to Figure 14 highlight the differences in node and edge coverage between structural and state drivers across clusters. Detailed descriptive statistics are provided in Table 53 of Appendix E. This table includes as well descriptive statistics on state drivers' RTP coverage. Results show that state drivers have slightly lower node and edge coverage than structural drivers. The difference, which is relatively small (as low as 1% in edge coverage for `Elevator` in Sannio 2), can simply be explained by the fact that participants working with code used node and edge coverage analysis to refine their test drivers and achieve better coverage.

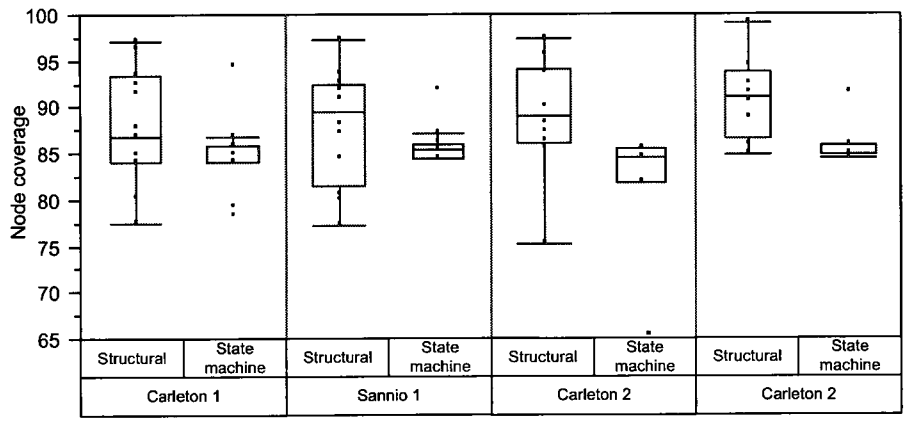


Figure 9: Node coverage distribution of Cruise Control's drivers

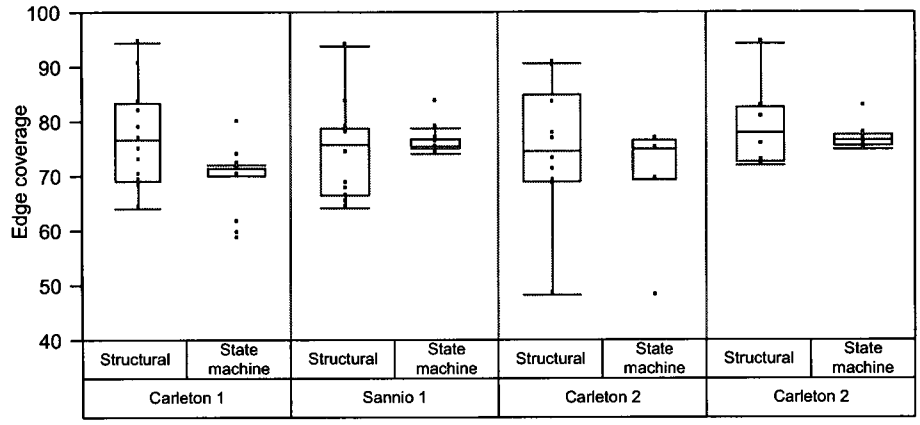


Figure 10: Edge coverage distribution of Cruise Control's drivers

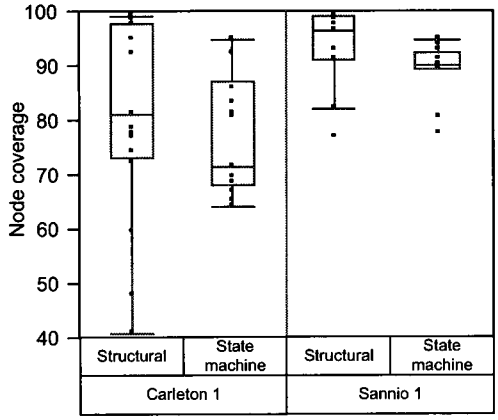


Figure 11: Node coverage distribution of OrSet drivers

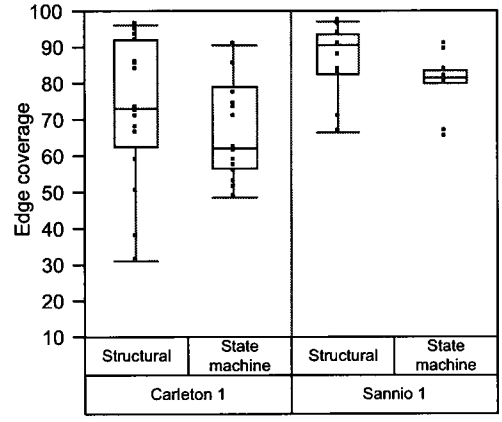


Figure 12: Edge coverage ditribution of OrdSet drivers

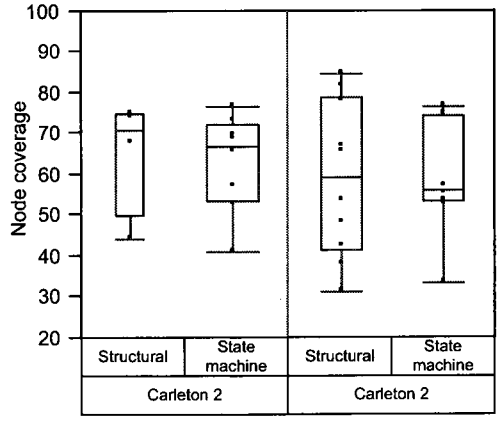


Figure 13: Node coverage distribution of Elevator drivers

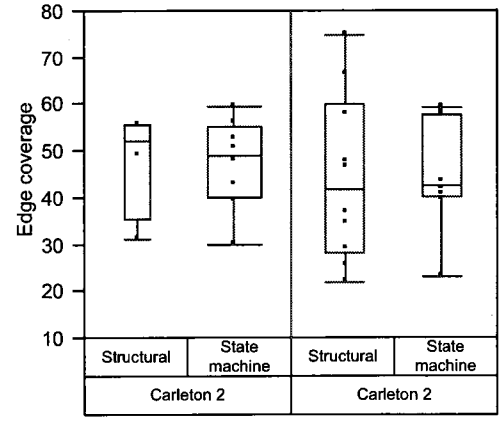


Figure 14: Edge coverage distribution of Elevator drivers

The only exception was edge coverage for `Cruise Control` in Sannio 1 where state drivers had a slightly higher coverage than structural drivers (the difference is less than 2%). Recall that for `Cruise Control`, state drivers show little variation in mutation scores because of similar RTP coverage. As the same paths (RTPs) are covered by state drivers, little variation in code coverage is also observed. This isn't the case for structural drivers where an important variation is observed in terms of mutation scores and code coverage, which can be explained by combined effect of a larger degree of freedom when coding structural drivers and variations in participants' ability. `Cruise Control`'s state drivers show better code coverage in the replications than in Carleton 1. In the replications, sneak paths were tested in additions to RTPs, thus covering larger areas of the code.

Clusters properties and functionalities have an important impact on the results of both mutation scores and code coverage. For `OrdSet`, the implemented functionalities are intuitive and clear from both its code and state machine. As a result, for both test techniques, mutation scores were definitely much higher for `OrdSet` than for `Cruise Control` or `Elevator` (Table 52). For node and edge coverage, results for `OrdSet` and `Cruise Control` were comparable, and they were much higher than those of `Elevator` regardless of the test technique (Table 53). `Elevator` has a significantly larger source code and given its real-time and concurrent behavior and state machine complexity, it is more difficult to achieve high coverage.

While `Cruise Control` exhibits real-time behavior, most changes to class attributes such as "speed", "distance" and "throttle" are computed in two methods (i.e., limited number of edges and nodes), which are the "run" methods of threads representing the car and its speed controller. Although these two methods are only of few lines of code (their coverage does not considerably affect node/edge coverage percentages), they include a significant number of computation and

assignment statements for which the number of generated mutants is high. Many of these statements were not covered by state drivers. In order to be covered, driver's execution time would need to be extended to allow for class attributes to change over time. State drivers did not take into account execution time as the real-time behavior of the cluster was not modeled by the state machine. As a result, a large number of mutants remained undetected.

b. Assessing the statistical significance of the difference in code coverage of drivers from both test techniques

For each cluster, and each experiment, we performed a t-test to assess the statistical significance of the difference in terms of node and edge coverage between the test techniques. We also performed t-tests on pooled data from all experiments. Statistical results are reported in Table 11 and Table 12.

Experiment	Cluster	DF	Node coverage mean		Effect size		t-value	Pr > |t|
			Structural	State machine	Observed	At 80% power		
Carleton 1	OrdSet	25.87	81.08	76.93	0.28	0.98	-0.81	0.425
	Cruise Control	22.72	87.92	84.79	0.63	1.03	-1.74	0.095
Sannio 1	OrdSet	17.45	93.12	89.11	0.62	1.23	-1.48	0.1561
	Cruise Control	13.44	87.97	85.75	0.48	1.20	-1.17	0.2609
Carleton 2	Elevator	5.13	65.04	62.81	0.17	1.90	-0.26	0.7993
	Cruise Control	12.02	89.18	81.19	1.17	1.48	-2.35	**0.0363**
Sannio 2	Elevator	16	58.75	56.85	0.12	1.32	-0.26	0.7961
	Cruise Control	8.82	90.92	86.04	1.38	1.38	-2.81	**0.0204**
All experiments	OrdSet	47.06	85.81	81.8	0.34	0.74	-1.11	0.2716
	Cruise Control	77.26	88.75	84.76	0.42	0.59	-3.81	**0.0003**
	Elevator	22.49	60.55	59.65	0.06	1.03	-0.16	0.8722

Table 11: t-test results for node coverage comparison across test techniques

For `OrdSet`, no statistically significant difference in code coverage (node and edge) was found between state and structural drivers. `OrdSet`'s state machine models closely its different

functionalities allowing state drivers to have good code coverage levels. However, a Wilcoxon test shows a significant difference in both node and edge coverage in Sannio 1. More laboratory time allowed participants working on structural drivers to reach out for uncovered nodes and edges in `OrdSet` code.

Experiment	Cluster	DF	Edge coverage mean		Effect size		t-value	Pr > |t|
			Structural	State machine	Observed	At 80% power		
Carleton 1	OrdSet	27.95	73.53	67.5	0.35	0.98	-1.03	0.3116
	Cruise Control	21.66	76.96	69.27	1.03	1.03	-2.66	**0.0091**
Sannio 1	OrdSet	18.55	86.44	80.29	0.70	1.23	-1.66	0.9431
	Cruise Control	12.76	74.57	76.42	0.28	1.20	0.69	0.4994
Carleton 2	Elevator	5.16	47.66	47.19	0.04	1.90	-0.07	0.9463
	Cruise Control	14.41	74.78	70.04	0.42	1.48	-0.86	0.4
Sannio 2	Elevator	14.67	44.16	43.4	0.05	1.32	-0.11	0.9106
	Cruise Control	7.84	79.13	77.18	0.35	1.38	-0.71	0.4987
All experiments	OrdSet	49.35	78.6	72.62	0.38	0.74	-1.44	0.154
	Cruise Control	76.53	76.22	72.97	0.40	0.59	-1.92	0.0578
	Elevator	20.86	45.16	45.18	0.00	1.03	0.005	0.9959

Table 12: t-test results for edge coverage comparison across test techniques

No significant difference in terms of code coverage was detected between `Elevator`'s structural and state drivers. The high complexity of both the source code and state machine resulted in limited code coverage.

When considering `Cruise Control`, different results for node and edge coverage were observed across experiments. Structural drivers achieved significantly better node coverage than state drivers in Carleton 2 and Sannio 2. This result is further confirmed when combining data from all experiments. Having access to the source code as well as reports on uncovered code helped participants working on structural drivers to have better code coverage in their drivers than in state drivers. However, for edge coverage, the difference is only found significant in

Carleton 1. Recall that sneak path testing was implemented in the replication experiments allowing for better edge coverage for state drivers.

Although the statistical tests show in general no statistical difference of code coverage between structural and state drivers, note that the minimum effect size calculated to reach a 80% power is much higher than the observed effect size in all but one case. Even when pooling data from all experiments, the calculated effect size matching 80% power is still high. Therefore, the lack of statistically significant results in Table 11 and Table 12 may be due to insufficient statistical power.

c. RTP coverage comparison across clusters

As for RTP coverage, Figure 15 shows the RTP coverage distribution of state drivers per cluster. For detailed descriptive statistics on RTP coverage per cluster and per experiment refer to Table 53 in Appendix E. The lowest RTP coverage was in `Elevator` for which the state machine complexity and the number of RTPs were higher than those of `OrdSet` and `Cruise Control`. Allowing more laboratory time in Sannio 2 led to an improvement in RTP coverage (up by 8%). However, the average number of covered RTPs was still low (38%). For `OrdSet`, RTP coverage was greatly improved after allowing more laboratory time (from an average of 35% in Carleton 1 to 78% in Sannio 1). For `Cruise Control`, with a simple state machine, almost all drivers had 100% RTP coverage.

For both `OrdSet` and `Elevator`, there is a significant learning curve for the initialization procedure of a test case because of the complex path conditions in the corresponding transition trees.

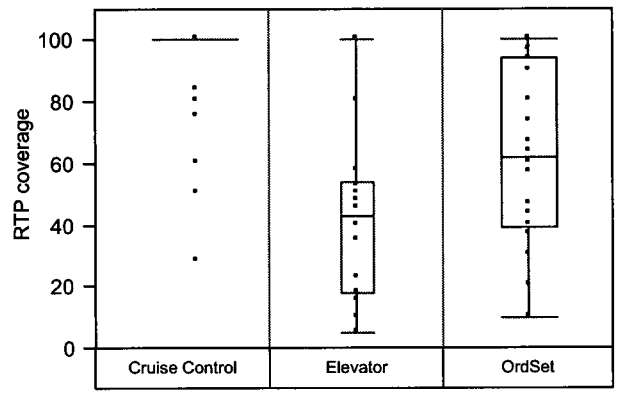


Figure 15: RTP coverage distribution for state drivers

For instance, in `OrdSet`, a test case initialization requires choosing sets of integers with specific characteristics such as the number of elements and the size of their intersection. In `Elevator`, the initialization of a test case requires choosing the number of elevators and floors in the cluster, the type of request, and the floor or elevator from which to send the request in order to ensure covering a specific path in the transition tree. Such test case initialization required time to be fully understood. As additional test cases are implemented, participants get increasingly familiar with the implementation of initialization procedures. Increasing laboratory time in Sannio 1 and 2 therefore allowed participants to achieve a better RTP coverage than in Carleton 1 and 2. Another source of variation in RTP coverage is participant ability. Different participants with different abilities covered different numbers of RTPs. This variation is clear for both `OrdSet` and `Elevator`. For `Cruise Control`, where the state machine was simpler, most participants covered all RTPs in their test cases.

The application of Holm's procedure to the pooled data from all experiments to account for possible type I error chance capitalization as described in Section 3.5 did not result in any change to the results already discussed in this section.

4.1.2 Interaction effects

Though our main interest is the relationship between test technique and cost effectiveness of testing, we expect this relationship to be affected by a number of interaction factors. In this section we focus on the interaction effects on mutation score between the test technique and the following factors: code coverage, cluster, laboratory order, and participant ability. A two-way Analysis of Variance (ANOVA) was performed to assess both the main and interaction effects involving test technique.

4.1.2.1 Interaction effects with code coverage

There is a strong linear correlation between node and edge coverage (R^2 = 0.9, 0.89, 0.93 and 0.97 for the four experiments, respectively) and, as a result, we limit any subsequent analysis to node coverage. ANOVA results showing the main and interaction effects of test technique with node coverage on mutation score are reported for different class clusters in Table 13, Table 14, and Table 15. For `Cruise Control`, the results show a significant main effect of the test technique and a significant interaction effect of node coverage with test technique. The only exception is the non-significant interaction effect in Carleton 2. This may be attributed to the relatively small number of observations (16) in Carleton 2. No significant main or interaction effect is observed for the other two clusters. Note, however, that when performing a two-way ANOVA of mutation score by test technique and node coverage without interaction, node coverage has a significant main effect on mutation score: As expected, higher code coverage corresponds to higher mutation scores.

Experiment	RSquare	Effect size	Factor	Parameter estimate	Sum of Squares	F Ratio	Prob > F
Carleton 1	0.15	0.18	Test technique	3.79	488.23	1.62	0.2120
			Node Cov * Test technique	0.42	1217.53	4.05	0.0529
Sannio 1	0.005	0.005	Test technique	-0.82	15.42	0.09	0.7632
			Node Cov * Test technique	-0.04	1.34	0.01	0.9292

Table 13: ANOVA - Impact of the test technique and its interaction with node coverage on mutation scores of OrdSet

Experiment	RSquare	Effect size[9]	Factor	Parameter estimates	Sum of Squares	F Ratio	Prob > F
Carleton 1	0.36	0.56	Test technique	2.29	162.21	4.71	**0.0387**
			Node Cov * Test technique	0.71	368.48	10.69	**0.0028**
Sannio 1	0.4	0.67	Test technique	4.5	487.04	4.79	**0.04**
			Node Cov * Test technique	1.42	955.47	9.41	**0.0059**
Carleton 2	0.47	0.89	Test technique	10.39	1579.74	10.59	**0.0063**
			Node Cov * Test technique	0.68	311.45	2.09	0.1721
Sannio 2	0.52	1.08	Test technique	5.73	580.27	8.1	**0.0123**
			Node Cov * Test technique	1.6	469.61	6.55	**0.0218**

Table 14: ANOVA - Impact of the test technique and its interaction with node coverage on mutation scores of Cruise Control

Experiment	RSquare	Effect size	Factor	Parameter estimate	Sum of Squares	F Ratio	Prob > F
Carleton 2	0.08	0.09	Test technique	2.58	67.49	0.68	0.4345
			Node Cov * Test technique	-0.05	4.19	0.04	0.8427
Sannio 2	0.18	0.22	Test technique	0.97	17.86	0.14	0.7165
			Node Cov * Test technique	0.31	437.87	3.35	0.0860

Table 15: ANOVA - Impact of the test technique and its interaction with node coverage on mutation scores of Elevator

[9] The effect size measure used in the context of a multiple regression is the Cohen's f^2[5].

Figure 16 shows interaction effect diagrams of test technique and node coverage for `Cruise Control` for experiments where the interaction effect was found significant. Similar behavior is observed in the three diagrams.

When code coverage increases, structural drivers tend to have higher mutation scores than state drivers. This can be explained by the fact that participants working with state machines cannot infer information related to real-time behavior that, as said, is not modeled by the state machines. Participants had no access to documentation on the real-time behavior, and therefore did not explicitly target real-time faults and certain parts of the code implementing the real-time behavior, i.e., implementation of the algorithm managing car speed and distance traversed over time and the algorithm managing the variation of the car throttle to maintain its speed when cruising is on. Even when fully covering the transition tree, one is unlikely to cover those parts of the code without precisely understanding the real-time properties. Those parts of the code represent a small percentage of the total number of nodes and edges, but a great number of mutants were created by seeding faults in them as they implement complex computations. However, for low code coverage, state drivers show higher mutation scores. This is likely due to the use of state invariant assertions in oracles. Note that the intersection between the interaction lines corresponding to structural testing and state testing occurs around 85% node coverage in all three graphs (node coverage average across experiments and test techniques - Table 53). This suggests that the structural testing technique starts to be more effective than state testing when the coverage is high (i.e., above 85%). For Sannio 2, Figure 16 – c, the interaction lines did not go below 85% of node coverage as this threshold corresponds to the minimum node coverage across drivers.

No interaction effect was found between node coverage and test technique for Carleton 2 though. This may be attributed to the relatively small number of observations (16) in Carleton 2 (of which six were state drivers and ten were structural drivers). Note, however, that when running ANOVA with test technique and node coverage without interaction variable, node coverage has a significant main effect on mutation score: As expected, higher code coverage corresponds to higher mutation scores. No significant main or interaction effect is observed for the other two clusters.

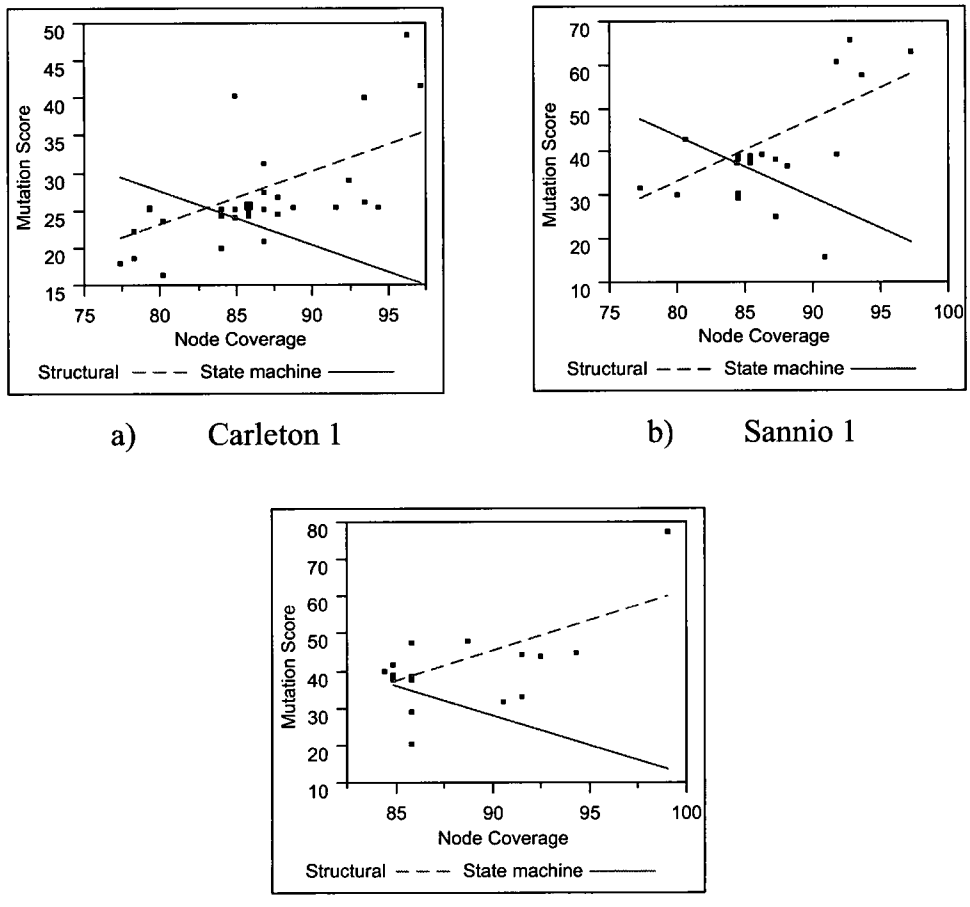


Figure 16: Test technique and node coverage interaction effect graphs - Cruise Control

4.1.2.2 Interaction effects with lab order

Lab order was the second factor for which we studied the interaction effect on mutation scores. This is to account for learning effects. It is assumed that participants would tend to perform better on the second lab than in the first lab, regardless of other factors. An analysis of variance (ANOVA) was performed and consistent results were observed across experiments. Results presented in Table 16, Table 17 and Table 18 report on the impact of test technique as well as its interaction with laboratory order on mutation score. The results showed no significant impact of lab order through interactions with test technique on mutation scores. A plausible reason is that participants were well trained for the tasks from the start and learning effects were therefore limited. Only one exception was observed in `Cruise Control` – Carleton 2, where a significant interaction effect was observed between lab order and test technique.

Experiment	RSquare	Effect size	Factor	Parameter estimate	Sum of Squares	F Ratio	Prob > F
Carleton 1	0.11	0.12	Test technique	2.31	165.4	3.49	0.0724
			Lab Order * Test technique	-0.39	4.78	0.1	0.7532
Sannio 1	0.21	0.27	Test technique	4.5	487.04	3.61	0.0711
			Lab Order * Test technique	6.75	258.37	1.92	0.1807
Carleton 2	0.55	1.22	Test technique	9.66	1400.91	11.18	**0.0053**
			Lab Order * Test technique	12.47	621.732	4.96	**0.0442**
Sannio 2	0.35	0.54	Test technique	6.2	683.8	7.13	**0.0174**
			Lab Order * Test technique	4.94	106.93	1.12	0.3076

Table 16: ANOVA - Impact of lab order and its interaction with test technique on mutation scores of Cruise Control

Figure 17 shows the interaction effect diagram of lab order and test technique in Carleton 2 – `Cruise Control`. Results for structural testing were better than of state testing in both labs, due to reasons discussed above (i.e., no modeling of the real-time behavior of `Cruise Control`). In addition, the interaction shows that structural testing results improved in the

second lab. This can be attributed to the increased knowledge gained by participants in-between the two labs (two weeks difference). But, unlike structural testing, state testing results in the second lab are clearly poorer than in the first one. The difference in knowledge of the test technique cannot be considered as a reason for the drop in mutation scores. Actually, participants did not practice on the RTP test technique any further after the first lab. However, this difference can be attributed to the small percentage of RTPs implemented on average in the second lab state drivers (17 RTPs out of 25 in lab 2, as opposed to 24 RTPs out of 25 in lab 1). Moreover, the low number of observations in Carleton 2 (6 state drivers for `Cruise Control` in total, 3 in first lab and 3 in the second) does not affect the general observation that no interaction exist between lab order and test technique .

Experiment	RSquare	Effect size	Factor	Parameter estimate	Sum of Squares	F Ratio	Prob > F
Carleton 1	0.05	0.05	Test technique	3.88	509.91	1.51	0.2282
			Lab Order * Test technique	-1.53	79.19	0.23	0.6314
Sannio 1	0.03	0.03	Test technique	-0.83	16	0.1	0.7558
			Lab Order * Test technique	3.96	86.81	0.54	0.4713

Table 17: ANOVA - Impact of lab order and its interaction with test technique on mutation scores of OrdSet

Experiment	RSquare	Effect size	Factor	Parameter estimate	Sum of Squares	F Ratio	Prob > F
Carleton 2	0.08	0.09	Test technique	2.54	65.38	0.65	0.4425
			Lab Order * Test technique	0.39	0.37	0.004	0.9532
Sannio 2	0.04	0.04	Test technique	0.96	17.5	0.11	0.7398
			Lab Order * Test technique	-4.08	78.44	0.51	0.4846

Table 18: ANOVA - Impact of lab order and its interaction with test technique on mutation scores of Elevator

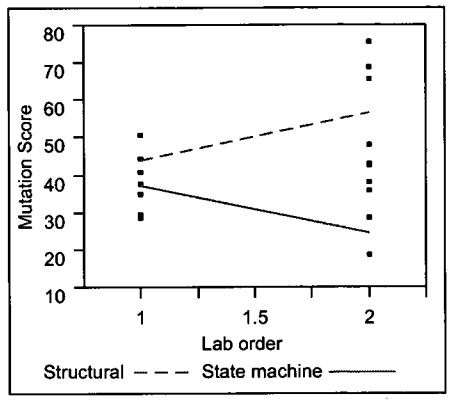


Figure 17: Test technique and Lab order interaction (Carleton 2 – Cruise Control)

4.1.2.3 Interaction effects with participant ability

When participant's ability increases, mutation score would a priori be expected to increase as well. The results of a two-way ANOVA with participant ability and its interaction with test technique on mutation scores are presented in Table 19, Table 20 and Table 21.

The tables report on the impact of test technique as well as its interaction with participant ability on mutation score. These results show that the interaction of participant ability and test technique has no significant effect on mutation score.

Experiment	RSquare	Effect size	Factor	Parameter estimate	Sum of Squares	F Ratio	Prob > F
Carleton 1	0.18	0.22	Test technique	2.32	167.18	3.81	0.0611
			S. Ability * Test technique	-3.67	104.02	2.37	0.1350
Sannio 1	0.14	0.16	Test technique	4.5	487.04	3.32	0.0828
			S. Ability * Test technique	-0.49	5.78	0.04	0.8446
Carleton 2	0.43	0.75	Test technique	9.33	1293.79	8.09	**0.0138**
			S. Ability * Test technique	-3.32	172.22	1.08	0.3182
Sannio 2	0.31	0.45	Test technique	6.17	677.73	6.59	**0.0214**
			S. Ability * Test technique	-0.86	2.86	0.03	0.8698

Table 19: ANOVA - Impact of participant ability and its interaction with test technique on mutation scores of Cruise Control

Experiment	RSquare	Effect size	Factor	Parameter estimate	Sum of Squares	F Ratio	Prob > F
Carleton 1	0.04	0.04	Test technique	3.79	488.23	1.44	0.2398
			S. Ability * Test technique	-0.35	1.06	0.003	0.9558
Sannio 1	0.01	0.01	Test technique	-0.94	20	0.12	0.7308
			S. Ability * Test technique	-0.92	19.01	0.12	0.7373

Table 20: ANOVA - Impact of participant ability and its interaction with test technique on mutation scores of OrdSet

Experiment	RSquare	Effect size	Factor	Parameter estimate	Sum of Squares	F Ratio	Prob > F
Carleton 2	0.12	0.14	Test technique	2.68	72.95	0.77	0.4070
			S. Ability * Test technique	-1.91	39.7	0.42	0.5366
Sannio 2	0.1	0.11	Test technique	0.99	18.6	0.13	0.7237
			S. Ability * Test technique	-7.64	230.98	1.61	0.2230

Table 21: ANOVA - Impact of participant ability and its interaction with test technique on mutation scores of Elevator

One possible problem is that participant ability as measured may not reflect properly testing skills, but rather the overall software engineering skills. Participants were mainly qualified into blocks based on their global grades or software engineering skills. A skill more specifically related to testing should have been employed instead to divide participants into appropriate blocks. It would have been probably more appropriate to evaluate participants in an introductory laboratory prior to experiment labs. The purpose of this evaluation would be to assess the testing knowledge gained by each participant prior to the experiment regarding the methods to be used for testing, but also general points such as motivation in participation in labs and precision in following instructions. Basili *et al.* [11] conjecture that one strategy for improving classroom experiments is to grade the participants on process conformance rather than results.

4.1.2.4 Interaction effects with class cluster

As for the last factor, class cluster, and its interaction with test technique on mutation scores, consistent results were observed across experiments. The results are presented in Table 22 which reports on the impact of test technique as well as its interaction with the tested cluster on mutation score. No significant interaction effect of cluster and test technique on mutation scores was observed.

Experiment	RSquare	Effect size	Factor	Parameter estimate	Sum of Squares	F Ratio	Prob > F
Carleton 1	0.03	0.03	Test technique	3.28	695.98	1.9	0.17
			Cluster* Test technique	-0.51	17.08	0.05	0.83
Sannio 1	0.03	0.03	Test technique	1.5	105.17	0.26	0.61
			Cluster * Test technique	3.01	424.92	1.06	0.31
Carleton 2	0.35	0.54	Test technique	6.43	1078.28	8.38	**0.01**
			Cluster * Test technique	3.48	295.4	2.29	0.14
Sannio 2	0.13	0.15	Test technique	3.37	418.76	3.33	0.08
			Cluster * Test technique	2.53	235.36	1.87	0.18

Table 22: ANOVA - Impact of cluster and its interaction with test technique on mutation scores

4.1.3 Summary

The main observation is that, when the state machine models the cluster behavior closely, no significant difference in terms of fault detection effectiveness is observed between state and structural drivers. However, when state machine modeling is inadequate, within practical boundaries, to capture key properties of the class cluster under test, then structural testing performs better. This is what was observed with `Cruise Control` where real-time properties could not be fully and adequately modeled.

Based on an analysis of interaction effects between test technique and a number of factors, only node coverage showed to be significant (structural drivers have higher mutation scores than state

drivers for higher coverage levels), and only in the case of `Cruise Control`. The most plausible reason is that, because of the lack of modeling of its real time properties, code coverage was limited for many drivers, thus creating more variation in coverage and making this interaction effect more visible than for other clusters.

High code coverage, in terms of nodes and edges, does not always indicate high fault detection ratio. Often a small, complex part of code, i.e., few nodes or edges, is a source for a large number of mutants. In practice too, we expect small, complex pieces of code to be more error-prone than others. By thoroughly exercising them the fault detection ratio may increase considerably while node and edge coverage may not increase much. In these cases data flow coverage or other, more fine-grained coverage criteria may be more appropriate, though they tend to be difficult to apply in practice due to the lack of appropriate, industry-strength tools.

4.2 Combining test techniques impact on fault detection effectiveness

To address RQ4, RQ5 and RQ6, we need to investigate whether it would be useful to complement test cases generated based on state machines (Ts) at design time, with those generated based on code coverage analysis (Tc).

When combining test cases, all pairs of drivers (state drivers × structural drivers) must be taken into consideration to capture the variability among drivers written by different participants. Having m state drivers and n structural drivers implies that $m \times n$ combinations would be considered. However, we do not consider drivers with low coverage, as we want to consider only realistic scenarios with competent developers and a reasonably disciplined process. Some drivers had very low coverage due to a combination of poor development skills, lack of compliance with the provided task instructions, and limited time. Therefore, in order to improve the external

validity of results to contexts with competent personnel and a reasonably disciplined process, we decided to compare only a subset of the drivers whose coverage was above a certain threshold, as described below.

However, though it is common practice to seek high statement coverage rates during testing in industrial test environments [81], this rate does not usually reach 100% due to budget and time restrictions, as well as the presence of unreachable code. Thus we chose 85% as a reasonable threshold for the selection of structural drivers in `OrdSet` and `Cruise Control`, and we lowered it to 65% in `Elevator` as only low coverage rates were achieved because of its complexity (Section 4.1.1.4).

As for RTP the decision is more complex as there is not much reported practice in state testing. We chose a 100% threshold for `Cruise Control` as only few participants did not cover all RTPs in their drivers. But for `OrdSet` and `Elevator`, very few participants were able to cover all RTPs or even achieve high RTP coverage rates (Table 53). In any case, we suspect that in a typical industrial environment, for a complex state machine with a large number of RTPs, only a subset of them is likely to fit within budgeted time and effort. Thus we selected a 60% RTP coverage threshold for `OrdSet` and 45% for `Elevator` so as to obtain a reasonably large subset of almost half-complete drivers.

Table 23 lists the number of selected and discarded drivers and the resulting total number of driver pairs that will be considered for the analysis in this section.

		# Selected		# Discarded		Total number of pairs to combine
		structural drivers	state drivers	structural drivers	state drivers	
Carleton 1	OrdSet	8	6	9	11	56
	Cruise Control	10	14	5	3	140
Sannio 1	OrdSet	9	12	2	1	99
	Cruise Control	9	11	3	1	99
Carleton 2	Elevator	3	3	1	4	9
	Cruise Control	9	3	1	3	27
Sannio 2	Elevator	5	4	5	5	20
	Cruise Control	7	10	1	0	80

Table 23: Drivers selection data for combining test techniques

To better understand how we chose a coverage threshold to select subsets of drivers, one must look at the line charts in the figures of Appendix D. These charts show the different plots for node, edge, and RTP coverage as well as the corresponding drivers' mutation scores per cluster, test technique, and experiment. Drivers are sorted by node coverage for the structural test technique and RTP coverage for the state test technique. As node coverage in structural drivers (respectively RTP coverage in state drivers) increases, mutation score increases as well. These thresholds represent a compromise between the level of completeness of the test drivers, that must be somewhat representative of the minimum expected coverage, and the resulting number of selected drivers that must be large enough to enable data analysis. Based on edge coverage for structural drivers and RTP coverage for state drivers, we identified the abovementioned criteria (thresholds) to select drivers for the remainder of this analysis.

4.2.1 Complementarity of testing techniques

In order to address question RQ4, we analyze the set of mutants killed by one type of drivers and not the other. How complementary is structural testing to state testing is captured by $|Fc - Fs|$ /

$|F|\%$[10]. Boxplots representing distributions of $|Fc - Fs| / |F|$ % are provided in Figure 19. In a realistic scenario, one would first generate state test cases, measure code coverage, and complement the test suite to achieve a certain level of structural coverage. The main motivation to follow that order is that generating large test suites being guided by code coverage analysis only is a highly tedious and time consuming task [65]. Selected descriptive statistics for all class clusters and experiments are reported in Appendix F.

Results show that on average the percentage of mutants killed only by structural drivers range from 14% to 27% for `Cruise Control`, 16% to 17.5% for `OrdSet`, and 8.4% to 11.5% for `Elevator`. The highest increase in mutation scores brought by structural drivers is for `Cruise Control`, for which real-time behavior related code was not fully covered by state drivers for the reasons already discussed in Section 4.1.1.1. The lowest increase is for `Elevator`, which can be considered the most complex in terms of code and state machine when compared to the other tested clusters. These results suggest that the contribution of structural drivers to a comprehensive global testing solution is significant but its extent depends on the complexity of the tested code. Edge and node coverage testing techniques contribute less when code complexity increases. When comparing the complexity, as measured by the number of edges and nodes (Table 1), of the three tested clusters, we note that `Cruise Control` is the cluster with less complexity and `Elevator` is the cluster with the highest complexity. The contribution of structural drivers to the global mutation score when combined with state drivers follow the same trend (Figure 18): The contribution was the highest for `Cruise Control` and the lowest for

[10] Recall from Section 3.2.3 that F is the set of all faults (mutants), Fs is the set of faults detected by a state driver and Fc is the set of faults detected by a structural driver (based on source code). $|Fc - Fs| / |F|\%$ is the percentage ratio of faults detected by a structural driver and not detected by a state driver out of the total number of faults.

`Elevator`. Other structural testing techniques, like data flow-based testing techniques, may improve the structural testing contribution, though this needs to be further investigated.

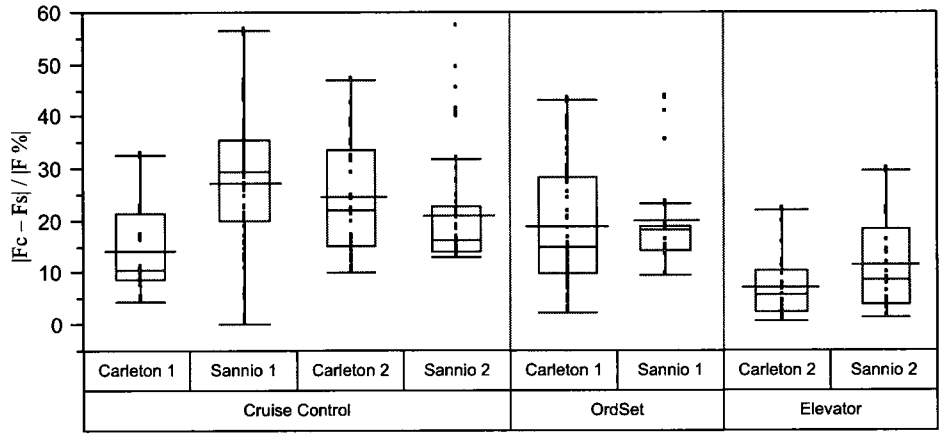


Figure 18: Distribution of |Fc - Fs| / |F| % across experiments and clusters

How complementary is state testing to structural testing is captured by $|Fs - Fc| / |F|$ %. Boxplots representing distributions of this measure are provided in Figure 17. Selected descriptive statistics for all class clusters and experiments are reported in Appendix F. For all experiments, the number of mutants killed *only* by state drivers represent on average a relatively modest percentage of all mutants of the cluster: it ranges from 6.6% to 8.4% for `Cruise Control`, 9.6% to 10.7% for `OrdSet`, and 10.4% to 14.4% for `Elevator`. However, this number sometimes reaches a considerably larger value (e.g., a maximum of 39% in `OrdSet`) which is probably not negligible in many practical circumstances where reliability requirements are high.

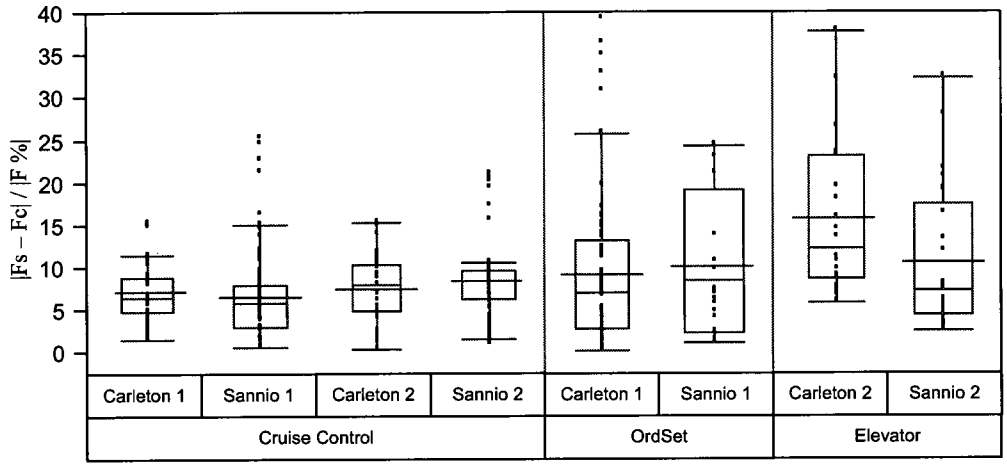


Figure 19: Distribution of |Fs - Fc| / |F| % across experiments and clusters

To further investigate the extent to which the two test techniques are complementary, we analyze the normalized intersection between the sets of detected faults by state and structural drivers $|Fs \cap Fc| / |F|$% and the mutation score proportion this intersection represents for each driver: $|Fs \cap Fc| / |Fs|$ and $|Fs \cap Fc| / |Fc|$. Such proportions determine the importance of the contribution of each type of driver to the overall mutation score resulting from combining testing techniques. Selected descriptive statistics are presented in Appendix F. Boxplots representing the distribution of the intersection and its ratios across experiments and clusters are provided in Figure 20, Figure 21, and Figure 22.

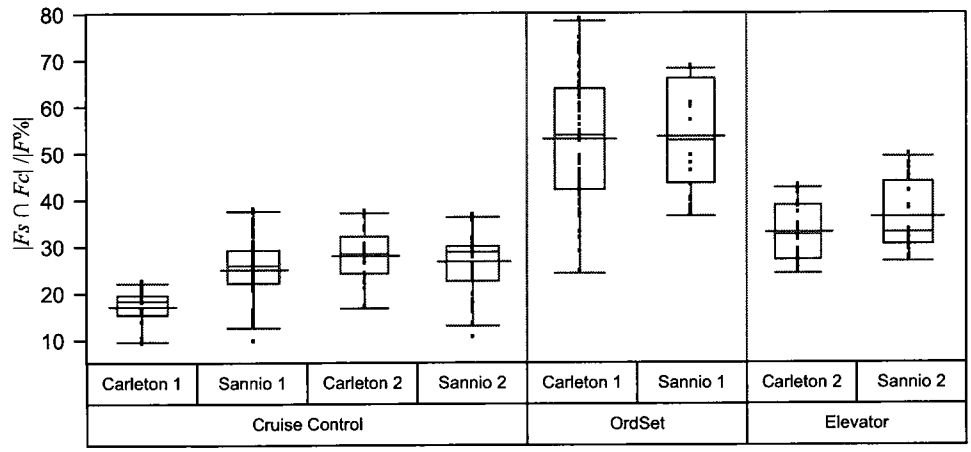


Figure 20: Distributions of the commonly detected faults by drivers from both techniques

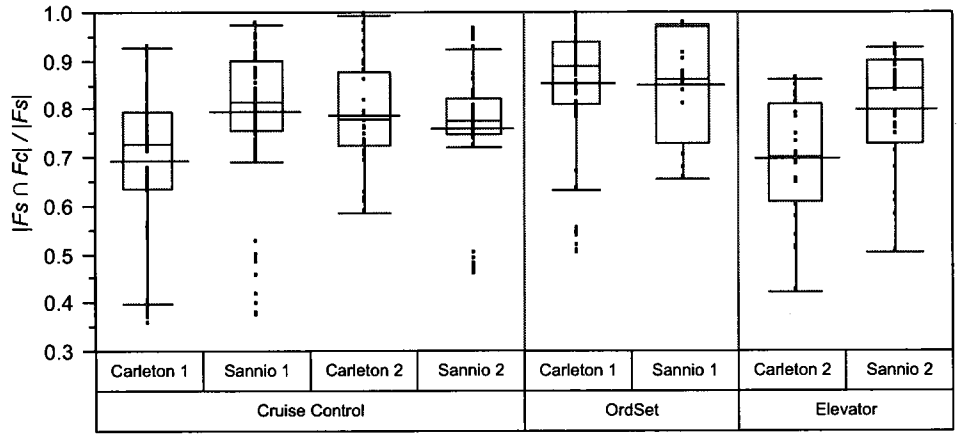


Figure 21: Distribution of |Fs ∩ Fc| / |Fs| ratio across experiments and clusters

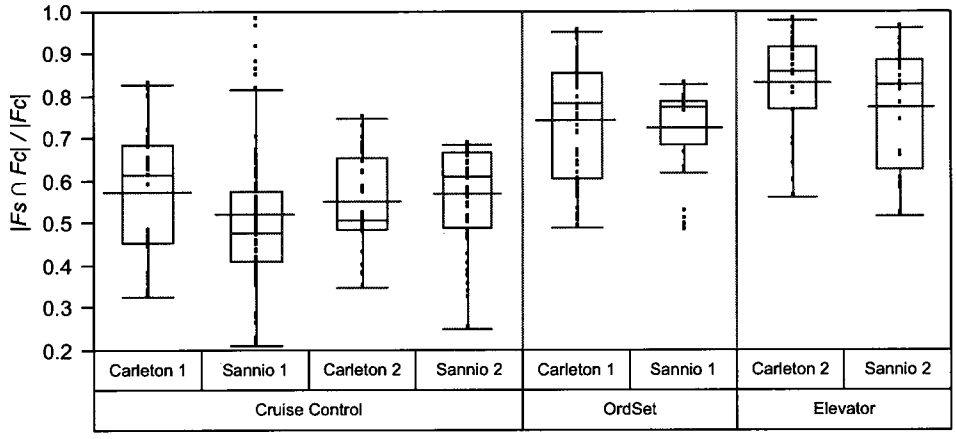


Figure 22: Distribution of |Fs ∩ Fc| / |Fc| ratio across experiments and clusters

Results were consistent across experiments and show that a practically significant proportion of faults were detected only by one type of drivers. For instance, the average of |Fs ∩ Fc| / |Fs| for `Cruise Control` varies from 0.69 to 0.79. This means that on average roughly 20% to 30% of the faults detected by state drivers are not detected by structural drivers. Similarly, on average around 40% of faults detected by structural drivers are not detected by state drivers. These ratios have lower average values for `OrdSet` where they range from 14% to 22% and for `Elevator` where they range from 13% to 20%. This difference between `OrdSet` and `Cruise Control` can be attributed to the fact that lower numbers of faults remained undetected in `OrdSet` and

therefore lower numbers of faults could be detected by only one type of drivers. For `Elevator`, the complexity of the code and state machine limited the number of faults detected by both techniques.

The results discussed so far further confirm that the two techniques are complementary in terms of fault detection and, as far as state testing is concerned, this leads to two questions: (1) why were the state test cases not able to detect faults detected by structural test cases? (2) Can the state testing technique be improved to detect faults that have shown to be detected only by structural drivers? An attempt to answer these questions is presented in Chapter 5 and Chapter 6.

4.2.2 Impact of combining test techniques on fault detection effectiveness

The following measures address the gain in terms of mutation score when combining test cases from both techniques:

- |Fs U Fc|/|F|% represents the mutation score in percentage when combining state and structural drivers.
- |Fs U Fc| / |Fs| represents a ratio measure of the gain in mutation score when combining drivers compared to state drivers alone.
- |Fs U Fc| / |Fc| represents a ratio measure of the gain in mutation score when combining drivers compared to structural drivers alone.

Selected descriptive statistics for these measures are presented in Appendix F. Corresponding distributions are shown in Figure 23, Figure 24, and Figure 25 below. As expected from the previous section, combining test cases from both test techniques clearly increased the mutation scores of combined drivers. The gain in mutation scores varied considerably across clusters and experiments. For instance, the mutation scores of state drivers was on average doubled in Sannio

1 / `Cruise Control`, while we only observe around 15% in average increase in Carleton 2 / `Elevator`. When combining techniques, high mutation scores were achieved for `OrdSet` (e.g., an average of 85% in Sannio 1), while lower rates were obtained for `Cruise control` (e.g., an average of 60% in Carleton 2) and for `Elevator` (e.g., an average 55% in Sannio 2). This, again, can be due to the already discussed real-time behavior and complexity of these clusters.

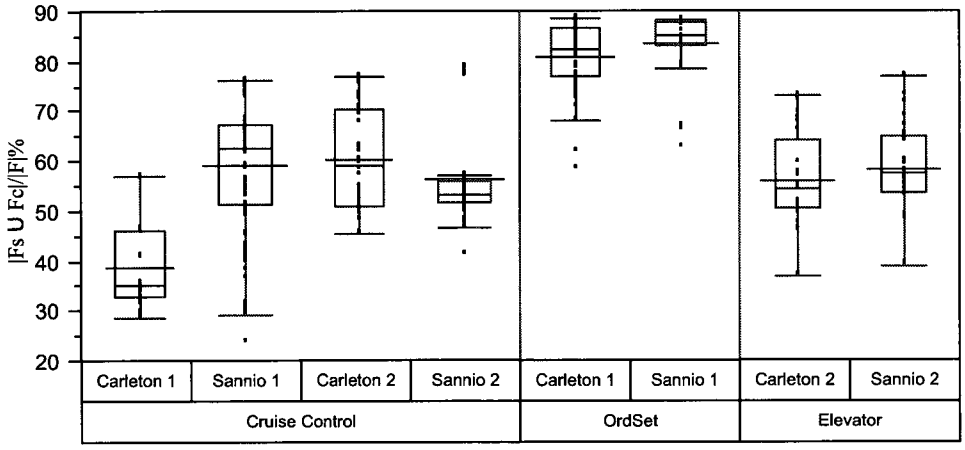


Figure 23: Distribution of |Fs U Fc| / |F| ratio across experiments and clusters

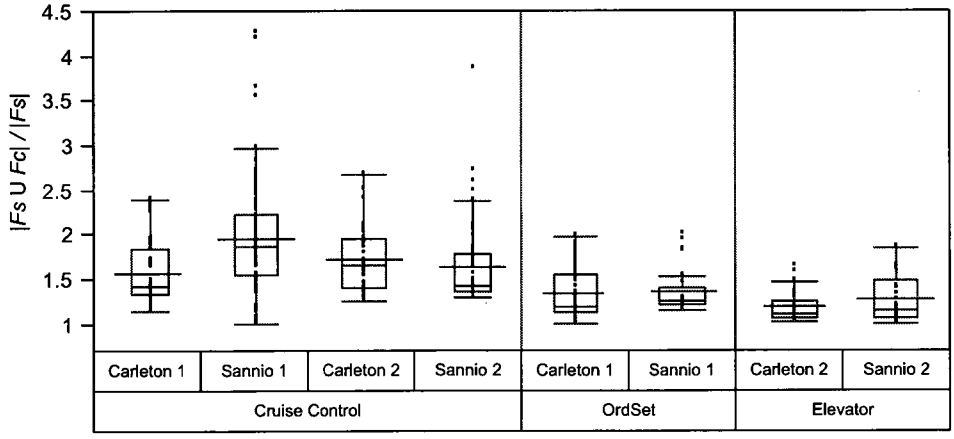


Figure 24: Distribution of |Fs U Fc| / |Fs| ratio across experiments and clusters

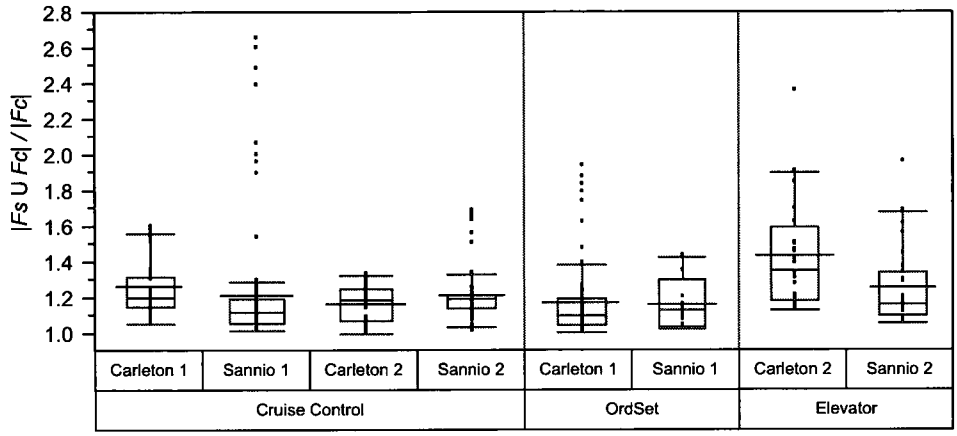


Figure 25: Distribution of |Fs U Fc| / |Fc| ratio across experiments and clusters

To formally address question RQ5, we need to analyze the significance of the gain in mutation scores when combining state and structural test cases. The following null hypothesis is tested: "*The fault detection rate when combining state testing and structural testing is equivalent to that obtained with structural testing alone and to that obtained with state testing alone*". Across the different clusters and experiments, 16 one-tailed *t*-tests for paired samples were performed to compare: (1) the means of mutation scores when including only structural test cases and after adding state test cases to them, and (2) the means of mutation scores when including only state test cases and after adding structural test cases to them. These tests are paired tests as each driver combination must be compared with its corresponding structural or state driver. Results of these paired, one tailed *t*-tests are shown in Table 24.

Results always indicate that the gain in mutation scores, from either structural testing or state testing, is statistically significant when combining test cases from the two testing techniques. The application of Holm's procedure to the pooled data from all experiments to account for possible type I error chance capitalization as described in Section 3.5 did not result in any change to the results presented and discussed in this section.

Therefore, when testing clusters with state-dependent behavior, it is recommended to adopt a testing approach in which state testing is first applied, and then complemented with structural testing.

	Cluster	Type of combination	DF	Mean of difference in mutation score	t value	Pr < t
Carleton 1	**Cruise Control**	State vs. combination	149	12.69	20.76	**<.0001**
		Structural vs. combination	149	7.28	25.00	**<.0001**
	OrdSet	State vs. combination	55	13.41	11.75	**<.0001**
		Structural vs. combination	55	9.67	7.64	**<.0001**
Sannio 1	**Cruise Control**	State vs. combination	98	18.78	18.60	**<.0001**
		Structural vs. combination	98	8.55	23.53	**<.0001**
	OrdSet	State vs. combination	142	13.89	15.17	**<.0001**
		Structural vs. combination	142	13.35	15.65	**<.0001**
Carleton 2	**Cruise Control**	State vs. combination	35	24.37	13.98	**<.0001**
		Structural vs. combination	35	7.39	11.19	**<.0001**
	Elevator	State vs. combination	11	5.11	2.26	**0.0447**
		Structural vs. combination	11	9.03	5.1	**0.0003**
Sannio 2	**Cruise Control**	State vs. combination	79	21.31	17.42	**<.0001**
		Structural vs. combination	79	8.14	13.42	**<.0001**
	Elevator	State vs. combination	24	8.28	4.24	**0.0003**
		Structural vs. combination	24	7.25	5.73	**<.0001**

Table 24: Combining test techniques - Paired *t*-tests results

In terms of practical significance, the improvements in mutation scores average between approximately 7% and 13% of all mutants across the three clusters when compared to structural testing alone alone and between 13% and 24% when compared to state testing. The improvements, for both test techniques, were higher in the replications for both `OrdSet` and `Cruise Control`. This indicates that the changes made to the experiment design (e.g., increasing lab time, adding contract assertions to oracles) improved not only the specific mutation scores of state drivers and structural drivers, but also their combined mutation scores.

4.2.3 Impact of cluster characteristics on combining test techniques

We performed *t*-tests to study the impact of cluster characteristics on the mutation score gain (RQ4) when combining test strategies. Table 25 reports on the gain in mutation scores when structural drivers are complemented with state drivers (|Fs – Fc|%), the gain in mutation scores when state drivers are complemented with structural drivers (|Fc – Fs|%), as well as the percentage of undetected faults by either type of drivers. In most cases, p-values show a statistically significant impact of clusters (i.e., their characteristics) on the gain in mutation scores and the number of faults undetected when combining structural and state drivers. Results for `OrdSet` are much better than those for `Cruise Control` and `Elevator` due to the already discussed code and state machine properties of the respective clusters.

		DF	Mean Cruise	Mean OrdSet	Mean Elevator	t value	Pr > |t|
Carleton 1	**Gain to structural mutation score |Fs-Fc|%**	57.24	4.51	9.63	-	-4.004	**0.0002**
	Gain to state mutation score |Fc-Fs|%	65.39	8.85	16.16	-	-5.087	**<.0001**
	Undetected faults |F-(FsUFc)|%	79.97	76.02	18.64	-	57.93	**0.0000**
Sannio 1	**Gain to structural mutation score |Fs-Fc|%**	191.72	9.16	13.35	-	4.52	**<.0001**
	Gain to state mutation score |Fc-Fs|%	295.8	19.35	13.88	-	-3.98	**<.0001**
	Undetected faults |F-(FsUFc)|%	223.77	45	16.34	-	-25.86	**0.0000**
Carleton 2	**Gain to structural mutation score |Fs-Fc|%**	14.02	7.47	-	9.03	0.83	0.4216
	Gain to state mutation score |Fc-Fs|%	40.37	24.5	-	6.88	-7.87	**<.0001**
	Undetected faults |F-(FsUFc)|%	29.49	40.02	-	46.26	2.49	**0.0188**
Sannio 2	**Gain to structural mutation score |Fs-Fc|%**	35.02	8.26	-	8.31	0.04	0.9705
	Gain to state mutation score |Fc-Fs|%	68.95	21.31	-	7.25	-7.99	**<.0001**
	Undetected faults |F-(FsUFc)|%	49.97	44.13	-	44.89	0.39	0.698

Table 25: Testing differences across clusters

4.2.4 Impact of combining test techniques on fault detection effectiveness across mutation operators

Results discussed in the previous sections show that state testing and structural testing are complementary in terms of faults detected. In this section we further investigate this question for each mutation operator (RQ6). This investigation aims at identifying fault types that are more likely to be detected by one test technique than the other. Table 26 to Table 29 list the total number of mutants created per mutation operator ($|F|$), the number of mutants killed only by state drivers ($|Fs - Fc|$), those only killed by structural drivers ($|Fc - Fs|$) and the total number of mutants killed when combining test cases from both drivers ($|Fs \cup Fc|$). Recall that the number of mutants per mutation operator changes with the cluster under test as this number is driven by the characteristics of the source code. Related descriptive statistics are presented in Appendix G.

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

Table 26: Count of detected and live mutants per mutation operator (Carleton 1)

Mutation operator	OrdSet				Cruise Control			
	$\|F\|$	$\|Fc-Fs\|$	$\|Fs-Fc\|$	$\|Fs \cup Fc\|$	$\|F\|$	$\|Fc-Fs\|$	$\|Fs-Fc\|$	$\|Fs \cup Fc\|$
JSI	5	2 (40%)	0	5 (100%)	11	1 (9%)	0	1 (9%)
JSD	3	0	0	0	9	0	0	0
JID					9	0	0	0
JDC					1	0	0	1 (100%)
EAM	4	0	0	3 (75%)	5	3 (60%)	0	3 (60%)
AORB	90	0	0	90 (100%)	28	28 (100%)	0	28 (100%)
AORS	8	0	0	8 (100%)				
AOIU	48	0	0	48 (100%)	31	13 (42%)	0	26 (84%)
AOIS	297	0	5(1.68%)	244 (82%)	148	56 (38%)	5 (3%)	131 (88%)
AODU	4	0	0	4 (100%)				
ROR	47	0	1(2.12%)	40 (85%)	79	16 (20%)	1 (1%)	51 (64%)
COR	6	0	0	6 (100%)				
COD	1	0	0	1 (100%)				
COI	4	0	0	4 (100%)				
LOI	107	1(0.93%)	1(0.93%)	106 (99%)	49	14 (28%)	0	43 (88%)
ASRS					12	4 (33%)	0	12 (100%)

Table 27: Count of detected and live mutants per mutation operator (Sannio 1)

Mutation operator	Elevator				Cruise Control			
	$\|F\|$	$\|Fc-Fs\|$	$\|Fs-Fc\|$	$\|Fs \cup Fc\|$	$\|F\|$	$\|Fc-Fs\|$	$\|Fs-Fc\|$	$\|Fs \cup Fc\|$
JTI	5	1	0	3				
JTD	4	1	0	3				
JSI	22	6	1	12	11	1	0	1
JSD	2	0	0	0	9	0	0	0
JID	6	0	0	3	9	0	0	0
JDC	1	0	0	1	1	0	0	1
EAM	56	9	3	22	5	5	0	5
AORB	76	3	13	24	28	28	0	28
AORS	6	0	0	6				
AOIU	88	4	12	68	31	14	0	27
AOIS	556	26	129	483	148	66	4	135
AODU	5	2	0	2				
ROR	107	5	21	64	79	21	0	53
COR	28	4	5	23				
COD								
COI	18	0	4	14				
LOI	192	11	36	142	49	18	0	46
ASRS	4	0	3	3	12	4	0	12

Table 28: Count of detected and live mutants per mutation operator (Carleton 2)

Mutation operator	Elevator				Cruise Control			
	$\|F\|$	$\|Fc-Fs\|$	$\|Fs-Fc\|$	$\|Fs \cup Fc\|$	$\|F\|$	$\|Fc-Fs\|$	$\|Fs-Fc\|$	$\|Fs \cup Fc\|$
JTI	5	1	0	3				
JTD	4	1	0	3				
JSI	22	9	0	16	11	1	0	1
JSD	2	0	0	0	9	2	0	2
JID	6	0	0	3	9	0	0	0
JDC	1	0	0	1	1	0	0	1
EAM	56	9	3	26	5	5	0	5
AORB	76	9	13	32	28	27	0	28
AORS	6	0	0	6				
AOIU	88	5	10	73	31	13	0	26
AOIS	556	42	74	508	148	58	4	137
AODU	5	2	0	3				
ROR	107	10	19	72	79	18	2	51
COR	28	6	2	25				
COD								
COI	18	4	4	17				
LOI	192	18	19	152	49	18	0	46
ASRS	4	1	0	4	12	4	0	12

Table 29: Count of detected and live mutants per mutation operator (Sannio 2)

A first trend we noticed when looking at sets of undetected faults is that the cluster under test seems to be an important factor that determines mutation operators for which state drivers are good detectors. For instance, out of the 90 AORB mutants created for `OrdSet`, 84 were killed by state drivers in Carleton 1 and all of them were killed by state drivers in Sannio 1. However, none of the AORB mutants created for `Cruise Control` has been killed by state drivers except one mutant killed in Sannio 2. For `Elevator`, state drivers detected up to 21 mutants out of the total of 76 AORB mutants. The AORB mutation operator replaces a binary operator such as the addition operator with another binary operator. In `Cruise Control`, such faults are seeded in the algorithm computing class attributes when the car is running (e.g., car speed, throttle). In order to detect such fault, a precise oracle is needed. However, it is extremely hard to know at a given point in time what would be the value of car speed for example. Such value depends on many factors: execution time, processor speed, and number of running processes on the CPU.

Therefore, oracles for `Cruise Control` cannot be very precise: attribute values can, in many cases, only be checked against an interval. For `Elevator`, despite its real-time characteristics, because we have a state machine that models precisely the behavior of the cluster and we can define relatively precise oracles, better results were obtained with AORB mutants than for `Cruise Control`. Moreover, for `OrdSet`, the characteristics of the class allow for very precise oracles: At any point in time, one can check class attribute values against exact expected values. To summarize, cluster characteristics have a strong impact on the precision of oracles, which in turn affect the effectiveness of test techniques.

Except for `OrdSet`, for which both structural drivers and state drivers detected almost all faults and of all types, different trends are observed for `Cruise Control` and `Elevator`. In `Cruise Control`, structural drivers overcame state drivers at detecting faults and for all types of faults. However, for `Elevator`, the trend is reversed. State drivers overcame structural drivers roughly for all fault types. This can be attributed as discussed above to cluster characteristics including structural and state machine complexities, and the precision of oracles. Following this observation, we investigate in Chapter 5 in a thorough and systematic manner the main causes for state drivers to fail at detecting faults and how cluster characteristics and other factors impact it.

4.2.5 Summary

Combining test cases from the two studied test techniques has a significant, positive impact on fault detection ratios and results are consistent across clusters and experiments. Across clusters and experiments, the highest average increase in mutation scores observed when state test cases were complemented with code test cases was 24% and, similarly, a 13% increase was observed

when code test cases were complemented with state test cases. In summary, the results suggest it is effective to combine test techniques to achieve better overall fault detection ratios, across different fault types. However, the improvement of fault detection ratios comes at the expense of an increased cost of the drivers (Section 4.3). A practical procedure would be to first apply state testing and then measure code coverage to augment the test suites.

An analysis of the faults detected by structural and state drivers per mutation operator showed that when combining drivers from both test techniques, the contribution of each type of driver to the detection of faults of some specific type depends on the cluster under test. It is therefore difficult to draw any firm conclusion for individual mutation operators.

4.3 Cost analysis of test techniques

In this section we address RQ7 by analyzing the difference between state testing and structural testing in terms of cost. This is complementary to the effectiveness analysis presented in the previous section as both differences in cost and effectiveness would have to be considered to select appropriate test strategies.

Testing cost can be generally divided into test generation cost and test execution cost. Regarding the former, we use as a surrogate measure the size of test drivers in terms of Lines Of Code (LOC) to compare the effort required for generating the test cases. Keeping testers' skills constant, such a measure assumes that the test generation cost is proportional to the driver's size. Test cases execution cost would be measured with two different measures: CPU time, measured in milliseconds (ms), provides an exact value for the time interval the CPU needed to execute the test cases. An alternative measure is the number of method calls, which focuses more on resources consumed—that are assumed to be proportional to the number of method calls—rather

than on the CPU usage. Though this is clearly a strong assumption, for practical reasons, it has been a common one in testing studies [8, 14, 25, 53, 91] where very often one test case corresponds to one execution of a function/program. This corresponds in our study to one execution of a cluster method. Given that most methods in object-oriented software are small, as the number of methods called grows, this count is likely to become a precise surrogate measure for test execution cost.

Table 30 below compares mean values of the three cost metrics, i.e. CPU time, LOC, and Number of method calls for state and structural drivers across clusters and experiments.

Cluster	Experiment	CPU time (ms)		LOC		Nb of method calls		Mutation score	
		Structural	State	Structural	State	Structural	State	Structural	State
OrdSet	Carleton 1	16	15	323	332	601	876	56.15	50.27
	Sannio 1	16	22	382	562	788	1732	70.31	71.96
Cruise Control	Carleton 1	8770	15	261	373	405	502	27.69	24.47
	Sannio 1	23460	16	188	716	247	1299	44.65	35.65
	Carleton 2	39186	16	281	618	684	1062	50.13	30.8
	Sannio 2	8342	16	275	1359	370	1104	46.89	34.55
Elevator	Carleton 2	33317	13466	240	511	455	867	40.54	35.44
	Sannio 2	32478	18266	255	550	729	1080	36.97	35.06

Table 30: Mean CPU time, LOC and number of method calls of collected drivers

When considering CPU time, different trends can be seen across clusters. For `OrdSet` which has no real-time behavior, as opposed to the other two clusters, both state and structural drivers had almost instantaneous execution time. For `Cruise Control`, where the real-time behavior was not modeled by its state machine, the difference in CPU execution time was enormous. While state drivers had almost instantaneous CPU execution time, structural drivers had very high CPU execution time (e.g. an average of 40 seconds in Carleton 2). This can be explained by the fact that participants working with the state machine had no information on the state-dependent behavior of the cluster while participants working with the code were in contact with the real-

time behavior of the cluster and noted the impact of the time factor on its attributes. A different trend was seen in `Elevator`. Recall that the real-time behavior of `Elevator` was modeled by its state machine and therefore it was accounted for in the application of the RTP technique. As opposed to the other clusters, state drivers had considerably high CPU execution time, but still much less than the measured CPU execution time for structural drivers. Participants working on structural drivers tended to focus more on analyzing real-time properties through the code and therefore put more emphasis on testing time-related properties, which was sometimes excessively tested with long periods of waiting times.

Comparing the test driver LOC, the results clearly show higher average cost for state drivers across all experiments and clusters. Similarly, the results of method calls show that state drivers tend to have a higher average cost than structural drivers. State drivers' cost increased in the replication experiments for both `OrdSet` and `Cruise Control`. All these observations lead to the following question: "*What are the factors that have an impact on the cost of state drivers and how can their impact be alleviated*?" An attempt to answer these questions is provided in Sections 5.4 and 7.2.

Two-sample *t*-tests were performed to obtain statistical evidence about the impact of test technique on the cost of test drivers (RQ7): Table 31, Table 32, and Table 33. Results show that the cost differences between the two test techniques are not consistently significant. We performed a non-parametric Wilcoxon test to ensure this was not due to a violation of the *t*-test assumptions.

Cluster	Experiment	DF	CPU execution time mean		CPU execution time variance		t value	Pr > |t|	Prob > |Z|
			Struct.	State	Struct.	State			
Carleton 1	**OrdSet**	32.54	15.52	15.33	0.26	0.23	-1.16	0.2551	0.2562
	Cruise Control	14	8769.67	15.24	732e6	0.19	-1.25	0.23	**0.0041**
Sannio 1	**OrdSet**	11.1	15.54	22.25	0.27	65.47	2.86	**0.0153**	**0.0263**
	Cruise Control	11	23459.8	15.8	519e6	0.15	-3.56	**0.0044**	**0.0004**
Carleton 2	**Elevator**	4.56	33316.5	21378.8	1347e6	674e6	-0.58	0.5883	0.6711
	Cruise Control	9	39185.7	15.7	2823e6	0.24	-2.33	**0.0447**	**0.0014**
Sannio 2	**Elevator**	12.11	32478	18266	3177e6	820e6	-0.73	0.4771	0.4869
	Cruise Control	7	8342	15.8	240e6	0.16	-1.52	0.1729	**0.0002**
All Experiments	**OrdSet**	29.43	15.53	18.1	0.26	36.85	2.3	**0.0284**	0.5334
	Cruise Control	44	19.370	15.6	1132e6	0.25	-3.86	**0.0004**	**<.0001**
	Elevator	18.61	32717.6	22899.3	2511e6	740e6	-0.66	0.5147	0.5684

Table 31: *t*-test results for cost analysis – CPU time

Experiment	Cluster	DF	LOC mean		LOC variance		t value	Pr > |t|	Prob > |Z|
			Struct.	State	Struct.	State			
Carleton 1	**OrdSet**	30.9	322.76	331.89	45841	30378	0.14	0.8913	0.8045
	Cruise Control	28.84	260.87	372.94	7831	15325	2.97	**0.0059**	**0.0055**
Sannio 1	**OrdSet**	19.4	382.36	562.08	24551	16471	2.99	**0.0073**	**0.0209**
	Cruise Control	12.65	187.5	715.58	2995	39685	8.85	**<.0001**	**<.0001**
Carleton 2	**Elevator**	7.13	239.75	691.5	1616	338124	2.18	0.643	0.0643
	Cruise Control	13.05	280.9	616.28	38613	38728	3.46	**0.0042**	**0.0072**
Sannio 2	**Elevator**	14.66	254.6	550.22	35902	255885	2.83	**0.0128**	**0.0128**
	Cruise Control	9.03	274.63	1358.5	6442	3455376	1.76	0.112	**0.0004**
All Experiments	**OrdSet**	55.75	346.18	423.97	37137	37211	1.53	0.1303	0.0652
	Cruise Control	47.45	248.2	711.28	13593	894787	3.33	**0.0017**	**<.0001**
	Elevator	20.87	250.36	691.11	25277	274696	3.37	**0.0029**	**0.0004**

Table 32: *t*-test results for cost analysis - LOC

Experiment	Cluster	DF	Nb of method calls mean		Nb of method calls variance		t value	Pr > |t|	Prob > |Z|
			Struct.	State	Struct.	State			
Carleton 1	**Cruise Control**	18.81	382.43	502.17	94084	14008	-1.453	0.1624	**0.036**
	OrdSet	25.98	600.64	875.55	167498	543753	-1.313	0.2004	0.6322
Sannio 1	**Cruise Control**	14.11	246.83	1298.50	15266	105845	10.468	**<.0001**	**<.0001**
	OrdSet	18.97	787.82	1732.33	180903	433470	4.119	**0.0006**	**0.0061**
Carleton 2	**Elevator**	6.84	455.25	867.43	352003	14814	1.77	0.1203	0.2696
	Cruise Control	13.84	684	1062.33	414568	171339	1.43	0.1749	**0.0453**
Sannio 2	**Elevator**	16.82	728.7	1079.78	641716	417342	1.06	0.3059	0.0757
	Cruise Control	15.7	369.5	1103.6	19904	42567	8.94	**<.0001**	**0.0003**
All Experiments	**OrdSet**	43.19	674.18	1218.27	174925	704679	3.15	**0.0029**	**0.049**
	Cruise Control	89.53	418.622	949.08	146254	184486	6.26	**<.0001**	**<.0001**
	Elevator	20.67	650.57	1755.78	464118	5338986	20.67	0.0681	**0.0275**

Table 33: *t*-test results for cost analysis - Number of method calls

For `OrdSet`, which has no real-time behavior, both state and structural drivers had almost instantaneous—and not significantly different—CPU execution time. The application of Holm's procedure (see Section 3.5) adjusted the p-value in Sannio 1 to 0.2, thus indicating that there is no significant difference, in terms of cost, between structural and state drivers. For `Cruise Control`, we found significant differences: while state drivers were almost instantaneous, it was not the case for structural drivers as they more thoroughly exercised the cluster real-time behavior. For `Elevator` drivers there is a difference between state and structural drivers, although not statistically significant. Different from `Cruise Control`, the real-time behavior was modeled in `Elevator`'s state machine and therefore it was accounted for by state drivers.

The results for method calls are not consistent across experiments and clusters. Significant differences in method calls were found for `Cruise Control`. This wasn't the case for `Elevator`. For `OrdSet` the results were not consistent across experiments. The lack of significance in some experiments can be due to the small size of our samples. However, when

combining data from all experiments, state drivers exhibit a cost significantly higher than structural drivers. An investigation of the high cost of state testing is presented in Section 5.4.

Comparing the test drivers' LOC, the results show that state drivers tend to have a higher cost than structural drivers, though this difference is consistently statistically significant for `Cruise Control` only. Also, state drivers' cost increased in the replication experiments for both `OrdSet` and `Cruise Control`. All these observations lead to the following question: "What are the factors that have an impact on the cost of state drivers?" An attempt to answer this question is provided in Section 5.4.

When assuming the drivers' cost to be proportional to the number of method calls, the results presented in this section showed that state testing is more expensive and less cost effective than structural testing. As we will see in Section 5.4, the main reason for the high cost of state drivers is the excessive use of precise oracles. The results discussed so far show a high test case generation cost for state drivers, thus suggesting it is important to investigate how we can make it more cost-effective. An attempt to answer this question is provided in Section 7.2.

4.4 Investigating the cost-effectiveness of combining test techniques

This section addresses research question RQ8. We analyze the impact of augmenting state testing with structural testing on cost and compare the improvement brought to fault detection ratios to the cost increase, an analysis referred to as cost-effectiveness.

The analysis of Section 4.2.4 showed that combining test cases from the two studied test techniques has a significant, positive impact on fault detection ratios. To better assess the effectiveness of such a combination as a test procedure, it is imperative to study its impact on cost and compare the improvement brought to fault detection ratios to the cost increase. For that

purpose, we need to analyze the significance of the increase in cost when combining state and structural test cases. The following null hypothesis is tested: "*The cost when combining state testing and structural testing is equivalent to that obtained with structural testing alone and to that obtained with state testing alone*". This hypothesis is tested for each cost measure (three times in total). Across the different clusters and experiments, 16 one-tailed *t*-tests for paired samples were performed to compare: (1) the means of structural drivers' cost and the means of the cost of the combination, and (2) the means of state drivers' cost and the means of the cost of the combination. These tests are paired tests as each driver combination must be compared with its corresponding structural or state driver. Results of these paired, one tailed *t*-tests are shown in Table 34, Table 35 and Table 36.

	Cluster	Type of combination	DF	Mean of difference in number of method calls	t value	Pr < t
Carleton 1	**Cruise Control**	State vs. combination	149	420.2	16.95	**<.0001**
		Structural vs. combination	149	531.4	81.23	**<.0001**
	OrdSet	State vs. combination	55	888.5	21.36	**<.0001**
		Structural vs. combination	55	1608.1	16.25	**<.0001**
Sannio 1	**Cruise Control**	State vs. combination	98	257.33	19.38	**<.0001**
		Structural vs. combination	98	1366.55	60.32	**<.0001**
	OrdSet	State vs. combination	107	909.33	26.02	**<.0001**
		Structural vs. combination	107	1842.1	33.92	**<.0001**
Carleton 2	**Cruise Control**	State vs. combination	35	741.11	7.09	**<.0001**
		Structural vs. combination	35	1387.5	52.97	**<.0001**
	Elevator	State vs. combination	11	504.67	23.5	**<.0001**
		Structural vs. combination	11	3065	3.62	**0.002**
Sannio 2	**Cruise Control**	State vs. combination	79	382.43	24.46	**<.0001**
		Structural vs. combination	79	1143.2	44.35	**<.0001**
	Elevator	State vs. combination	24	1165	6.63	**<.0001**
		Structural vs. combination	24	2862.6	5.47	**<.0001**

Table 34: Comparing test techniques combination cost (Number of method calls) - Paired *t*-tests results

	Cluster	Type of combination	DF	Mean of difference in LOC	t value	Pr < t
Carleton 1	Cruise Control	State vs. combination	149	253.9	35.68	<.0001
		Structural vs. combination	149	389.53	40.87	<.0001
	OrdSet	State vs. combination	55	460	17.32	<.0001
		Structural vs. combination	55	504	26.11	<.0001
Sannio 1	Cruise Control	State vs. combination	98	195.67	35.04	<.0001
		Structural vs. combination	98	747.91	44.93	<.0001
	OrdSet	State vs. combination	107	428.33	33.89	<.0001
		Structural vs. combination	107	577.55	49.02	<.0001
Carleton 2	Cruise Control	State vs. combination	35	289.44	8.79	<.0001
		Structural vs. combination	35	657.5	54.05	<.0001
	Elevator	State vs. combination	11	245.67	21.2	<.0001
		Structural vs. combination	11	1094.5	7.07	<.0001
Sannio 2	Cruise Control	State vs. combination	79	287.57	35.09	<.0001
		Structural vs. combination	79	1289.9	6.34	<.0001
	Elevator	State vs. combination	24	328.2	7.35	<.0001
		Structural vs. combination	24	975.2	9.18	<.0001

Table 35: Comparing test techniques combination cost (LOC) - Paired *t*-tests results

	Cluster	Type of combination	DF	Mean of difference in CPU execution time	t value	Pr < t
Carleton 1	Cruise Control	State vs. combination	149	12346.8	4.8	<.0001
		Structural vs. combination	149	15.2	463	<.0001
	OrdSet	State vs. combination	55	15.5	229.9	<.0001
		Structural vs. combination	55	15.28	250.93	<.0001
Sannio 1	Cruise Control	State vs. combination	155	25299	12.41	<.0001
		Structural vs. combination	155	15.81	406	<.0001
	OrdSet	State vs. combination	107	15.55	309.9	<.0001
		Structural vs. combination	107	22.9	29.21	<.0001
Carleton 2	Cruise Control	State vs. combination	35	43538	5.01	<.0001
		Structural vs. combination	35	16	405	<.0001
	Elevator	State vs. combination	11	44318	5	0.0002
		Structural vs. combination	11	7039.78	5.55	<.0001
Sannio 2	Cruise Control	State vs. combination	79	9145	5.19	<.0001
		Structural vs. combination	79	15.8	357.5	<.0001
	Elevator	State vs. combination	24	64384	5.19	<.0001
		Structural vs. combination	24	43472	8.16	<.0001

Table 36: Comparing test techniques combination cost (CPU execution time) - Paired *t*-tests results

Results are consistent across experiments and cost measures and show that the increase in drivers' cost, from either structural testing or state testing, is statistically significant when combining test cases from the two testing techniques.

It is important to recall that the combination of test cases from both techniques was done by simply adding all test cases from one test technique driver to a driver from the other test technique, without removing potential redundant (in terms of coverage) test cases. Therefore, the cost values for combined drivers presented in this section represent the uppermost possible values where no redundant test cases are eliminated from the combination driver. In practice, one test technique would be used to generate a first set of test cases, and the other test technique would be used to complement the first by generating new test cases that are not redundant and complement code coverage. Below we present cost results of combined drivers after eliminating reduncant test cases.

In order to assess the effectiveness of combining the two test techniques, it is imperative to compare the increase in mutation scores achieved to the increase in drivers' cost. Table 37 below provides for both test techniques the mean values of mutation score increase and cost increase for all three cost measures across experiments and clusters. Figure 26, Figure 27 and Figure 28 visualize the increase in mutation scores and cost and compare driver combination to structural and state drivers' results. Recall that these values are calculated from a subset of the collected drivers as discussed in Section 4.2. The increase in CPU execution time varied considerably from 15 ms to over 64000 ms and therefore it is hard to graphically compare the increase in CPU execution time across experiments and clusters as we do for the other cost measures in Figure 27 and Figure 28.

Experiment	Cluster	Increase in mutation scores		Increase in drivers' cost					
				Number of method calls		LOC		CPU execution time	
		Structural	State	Structural	State	Structural	State	Structural	State
Carleton 1	Cruise Control	7.28	14.03	531.4	420.2	389.53	253.9	15.2	12346.8
	OrdSet	9.67	16.16	1608.14	888.5	504	460	15.29	15.5
Sannio 1	Cruise Control	6.55	27.05	1298.5	257.33	715.58	195.67	15.83	25298.7
	OrdSet	12.44	11.11	1842.09	909.33	577.55	428.33	22.91	15.56
Carleton 2	Cruise Control	7.39	24.37	741.11	657.5	657.5	289.44	24.37	43537.9
	Elevator	13.38	9.46	3065	504.67	1094.5	245.67	39121	44317.7
Sannio 2	Cruise Control	8.31	21.02	1143.18	382.43	1289.91	287.57	15.82	9145
	Elevator	10.45	11.48	2862.60	1165	975.2	328.20	43471.8	64384.2

Table 37: Observed increase in drivers' mutation scores and cost

While the increase in mutation scores from state drivers (ranging from 14% to 27%) was noticeably higher than from structural drivers in `Cruise Control` (ranging from 6.5% to 8%), it was comparable in the other two clusters. However, the increase in cost (for Number of method calls and LOC metrics) was clearly much higher for structural drivers across experiments and clusters. For instance, the LOC for structural drivers in Elevator/Sannio 2 increased by 975 compared to 328 for state drivers. When considering the CPU execution time metric, the increase in cost for `OrdSet` and `Elevator` drivers was comparable across testing techniques, while a large difference across techniques was found for `Cruise Control`. While `Cruise control`'s structural drivers addressed the real-time behavior of the cluster, the state drivers did not address it because this aspect was not modeled in the state machine. From the above results regarding LOC and number of method calls and the increase observed in mutation scores when combining drivers from both techniques, we can conclude that it is more cost effective to complement state drivers with test cases from structural drivers than doing the opposite. While the increase in mutation score is comparable between test techniques, the increase in cost is much lower in state

drivers complemented with structural test cases. As discussed previously, this is also more logical and practical as models are provided before source code is completed and therefore, software testers can start generating test cases based on the state test technique and complement their drivers later thanks to structural coverage analysis.

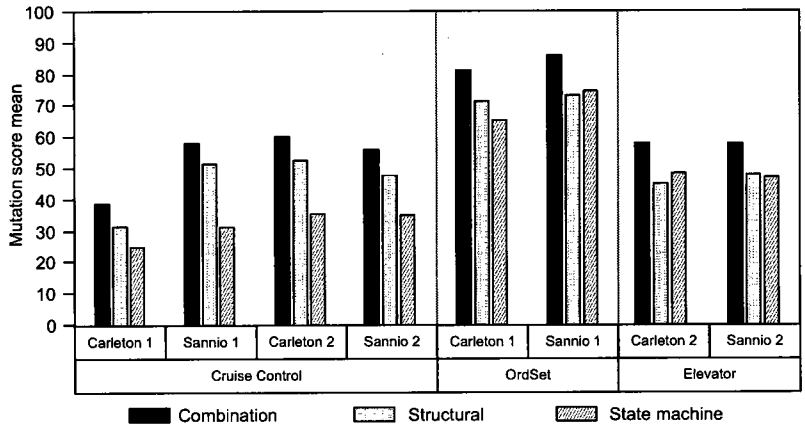


Figure 26: Mutation scores for driver combination, structural and state drivers

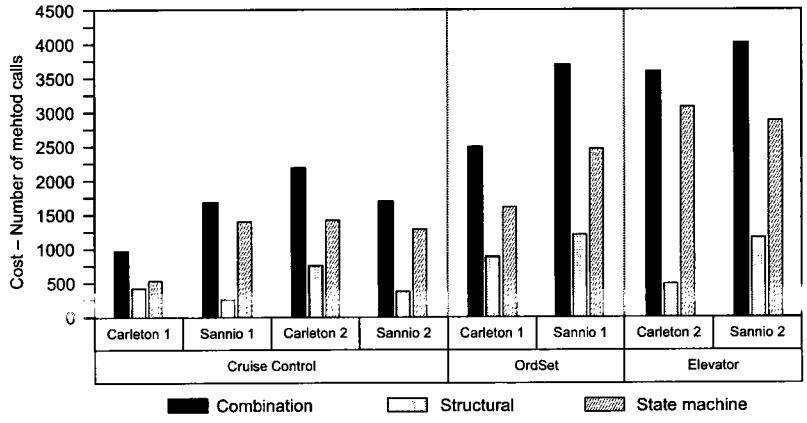


Figure 27: Method calls for driver combination, structural and state drivers

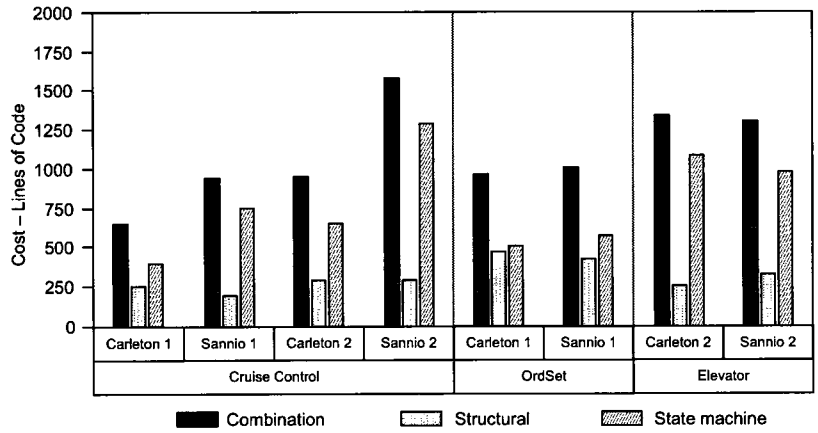


Figure 28: Cost (LOC) for driver combination, structural and state drivers

Simply combining drivers by merging all test cases from both types of drivers may lead to erroneous conclusions regarding cost. The combined drivers may include a significant number of redundant test cases in terms of code coverage. In practice, a test driver would be first generated based on the state machine of the cluster under test. Next, when the code becomes available, based on coverage analysis, the state driver would be complemented to improve code coverage with structural test cases, keeping coverage redundancy among test cases to a minimum. The analysis in this section uses some of the drivers developed in this experiment to emulate and assess this procedure in terms of its impact on mutation score and cost.

Combining test drivers while eliminating redundant test cases has been performed as follows. For each pair of drivers, `Ds` and `Dc` (state and structural drivers), we start by combining all test cases into one driver `Dsc`, for which the node and edge coverage are calculated (`Nsc` and `Esc`). `Nsc` and `Esc` represent the highest code coverage that can be achieved by augmenting `Ds` with `Dc` (i.e., state machine testing with structural testing). However, this code coverage can perhaps be achieved with a subset of non-redundant (in terms of code coverage) test cases from `Dsc`. To remove any potential redundant test case from the augmented driver `Dsc`, we proceed by removing one structural test case (`Dc`) at a time. If the code coverage measured after removing

the test case does not change (from Nsc and Esc), then we consider the removed test case as a redundant test case, which should be eliminated from the augmented driver Dsc. Instead, if the code coverage drops, the test case should be kept in the augmented driver Dsc. We repeat this procedure for all structural test cases in Dc. This results in an augmented driver with no redundant test cases and the highest possible code coverage given available structural test cases. The procedure presented above is intended to be representative of situations where RTP is adequately applied and code coverage is maximized within time constraints.

To address RQ8 using the test drivers developed in the experiment, the analysis proceeded by selecting three complete state test drivers (one for each cluster under test), implementing all RTPs with contract checking in oracle assertions. Similarly, we selected for each cluster three structural drivers to be used for augmenting the state driver among those collected drivers having the highest code coverage. Recall that our goal is to emulate the process of augmenting state drivers with structural test cases based on coverage analysis, in order to maximize coverage based on realistic test cases and within time constraints. The selected, high coverage drivers were generated by participants within the experiment time constraints and provide more opportunities (than other structural drivers) of finding test cases to augment and maximize coverage. The reason for selecting three structural test drivers for each cluster is to account for the significant variation observed among these drivers in terms of test cases.

Let us look first at the impact on mutation scores of using the procedure described above to form test drivers. Let Ms be the mutation score of state drivers and Msc the mutation score achieved by augmented drivers (i.e., after eliminating redundant test cases). Table 38 presents mutation score results of augmented drivers and their relative increase in mutation scores, i.e., ΔMs = (Msc -

Ms) / Ms (when compared to state drivers), and ΔMc = (Msc - Mc) / Mc (when compared with structural drivers).

Cluster	Driver	Mutation scores			Relative increase of mutation scores	
		State	Structural	Combination	ΔMs	ΔMc
Cruise Control	Driver 1	39.53	61.58	63.6	0.61	0.03
	Driver 2	39.53	68.32	71.99	0.82	0.05
	Driver 3	39.53	75.13	76.96	0.95	0.02
OrdSet	Driver 4	79.17	71.96	86.7	0.10	0.20
	Driver 5	79.17	78.85	88.3	0.12	0.12
	Driver 6	79.17	83.33	86.7	0.10	0.04
Elevator	Driver 7	65.14	60.71	73.97	0.14	0.22
	Driver 8	65.14	49.83	70.41	0.08	0.41
	Driver 9	65.14	53.66	72.5	0.11	0.35

Table 38: Mutation scores of augmented drivers

The relative increase in mutation scores of augmented drivers with respect to state drivers is the highest in Cruise Control (increase up to 95%), while it is much lower for OrdSet (increase up to 12%) and Elevator (increase up to 14%), for which state drivers had high mutation scores. The relative increase in mutation scores of augmented drivers with respect to structural drivers is the highest in Elevator (increase up to 41%), while it is much lower for Cruise Control (increase up to 5%).

Similar to mutation scores, for each cost metric, we compute the increase in cost due to augmenting drivers relative to the original cost of state and structural drivers. For example, let Cs be the cost of state drivers, Cc the cost of structural drivers, and Csc the cost of augmented drivers, then the relative increase in cost is computed as ΔCs = (Csc - Cs) / Cs (with respect to state drivers), and ΔCc = (Csc - Cc) / Cc (with respect to structural drivers). Finally, we define the *cost-effectiveness* of augmenting state with structural test cases as the ratios ΔMs/ΔCs and ΔMc/ΔCc, considering the cost-effectiveness of augmented drivers with

respect to state and structural drivers respectively. The motivation is to compare the increase in mutation score to the increase in cost. For example, a cost-effectiveness of one tells us the relative increases in mutation score and cost are equal. A value below one suggests the increase in mutation score is smaller than that of cost and a value above one suggests the contrary. Table 39 shows cost-effectiveness results for state and structural drivers for the three cost metrics.

Cluster	Driver	Cost-effectiveness					
		CPU execution time		Number of method calls		LOC	
		ΔMs/ΔCs	ΔMc/ΔCc	ΔMs/ΔCs	ΔMc/ΔCc	ΔMs/ΔCs	ΔMc/ΔCc
Cruise Control	Driver 1	<0.01	-0.08	0.48	0.05	1.40	0.02
	Driver 2	<0.01	-0.07	2.30	0.04	2.43	0.13
	Driver 3	<0.01	-0.03	2.32	0.02	2.36	0.10
OrdSet	Driver 4	0.14	0.31	0.27	0.11	0.18	0.68
	Driver 5	0.16	0.19	0.75	0.02	0.67	0.05
	Driver 6	0.16	0.08	0.26	0.02	0.28	0.06
Elevator	Driver 7	0.15	0.93	1.08	0.09	0.76	0.24
	Driver 8	0.44	0.20	2.67	0.04	0.84	0.08
	Driver 9	0.06	0.90	1.06	0.07	0.40	0.11

Table 39: Cost-effectiveness for augmented drivers with respect to state and structural drivers

For `Cruise Control`, the cost-effectiveness of augmented drivers is higher with respect to state drivers when considering the cost metrics "Number of method calls" or "Lines of Code" (gray cells). The gain in mutation score is much larger than the increase in cost. This is not the case with respect to structural drivers where the increase in cost is larger than the gain in mutation score. This can be explained by the fact that `Cruise Control`'s state drivers had limited code coverage as the real-time behavior of the cluster was not modeled in the state machine. When considering the "CPU execution time metric", results are different from the other two metrics. Though the gain in mutation scores of state drivers was high (Table 38), the cost increased so much that the cost-effectiveness of adding structural test cases to a state driver was close to 0. With respect to structural drivers, the cost effectiveness is even negative. This is

because augmented drivers led to a slight increase in mutation score (Table 38), while they exhibited a decrease in CPU execution time with respect to structural drivers. This is because augmented drivers did not contain the structural test cases which were redundant with respect to the chosen state drivers and particularly expensive in terms of CPU time.

For `OrdSet`, the cost-effectiveness of combining drivers is in general low. The gain in mutation score was small compared to the increase in cost, regardless of the specific drivers and cost metrics used. On one hand, `OrdSet`'s state machine fully modeled the functionalities of the cluster leading to high mutation scores and high coverage in state drivers when covering all RTPs, and on the other hand, the rather intuitive functionality implemented in source code of one class led to high mutation scores and high coverage in structural drivers. Therefore the increase in mutation scores when augmenting state drivers was smaller than the cost increase to achieve higher coverage.

For `Elevator`, the cost-effectiveness of augmented drivers with respect to state drivers (ΔMs/ΔCs) was only high with respect to "Number of method calls". The increase in state drivers' mutation scores was related to covering specific functionalities of the cluster not modeled by the state machine (mainly concurrency). With dedicated test cases covering parts of the code implementing concurrency, structural drivers led to an increase in code coverage and mutation score while incurring a relatively small increase in cost.

We may not be able to generalize the results of this section, but we can identify a trend: the cost-effectiveness of complementing state drivers with structural test cases is relatively high when the state machine modeling a cluster under test does not fully model its functionalities. This is rather the general case as state machines are rarely used to model entire systems (further discussions of this point can be found in Sections 3.1 and 6.4). Structural test cases covering code implementing

the non-modeled functionality would bring an increase in mutation score of the augmented driver with relatively low cost.

Combining test cases from the two test techniques proved to significantly improve fault detection effectiveness but at the expense of increasing test generation and execution cost. However, the increase in testing cost can be limited by eliminating redundant test cases in terms of node and edge coverage. It is important to investigate ways for lowering state drivers's cost without lowering their fault detection effectiveness. This is addressed in Section 7.2 where we propose changes to the state testing strategy in order to improve its cost effectiveness.

4.5 Equivalent mutants' analysis

Equivalent mutants are mutants that always produce the same output as the original program. In any application of mutation testing, one should expect a percentage of equivalent mutants preventing test drivers from achieving a 100% mutation score. Statistics show that equivalent mutants are not uniformly distributed among the mutants' types and they tend to cluster among a few types [77].

Finding equivalent mutants manually is a hard task especially for large programs as they tend to generate great numbers of mutants. A number of studies discussed and proposed techniques and heuristics for detecting equivalent mutants [10, 75, 77]. Empirical studies addressing those techniques show that only subsets of the equivalent mutants are typically detected. In many cases, equivalence is an undecidable problem.

While processing the data collected from the different experiments, we analyzed all undetected (live) mutants for the purpose of understanding the limitations of test techniques, with a particular focus on state drivers (Section 5.1). For each live mutant by state drivers, we created

execution traces for the faulty program to understand why some faults were not detected. This helped us identify equivalent mutants that do not change programs outputs and behavior. For instance, a mutant that deletes an initialization of an integer to zero is an equivalent mutant as the compiler would automatically assign an initial value of zero to uninitialized attributes. Other types of mutants are more complex and may for example require studying boundaries of attributes in some loops.

In total, 33 mutants were found to be equivalent mutants in `Cruise Control`, 64 in `OrdSet` and 121 in `Elevator`. This corresponds to 8.5 %, 10% and 10% of the total generated mutants in the three clusters, respectively. Table 40 provides details on the percentage of live mutants after executing all drivers from the total number of generated mutants and the ratio of equivalent mutants out of the live mutants for each treatment.

Experiment	Cluster	|F – Fc|%	|F – Fs|%	|Ne|/|F – Fc|	|Ne|/|F – Fs|
Carleton 1	Cruise Control	28.5	74.8	30.3	11.5
	OrdSet	13.5	16.5	76.2	62.1
Sannio 1	Cruise Control	24.1	57.8	35.9	14.9
	OrdSet	11.5	10.9	88.9	94.1
Carleton 2	Cruise Control	20.4	60.4	42.3	14.3
	Elevator	45.1	34.9	22.8	29.5
Sannio 2	Cruise Control	20.7	57.3	41.8	15.1
	Elevator	32	29.7	32.2	34.7

Table 40: Count of live mutants per experiment and per treatment

We consider live mutants those still undetected when running all the available state drivers and structural drivers. $|F - Fc|$% represents the percentage of live mutants corresponding to faults undetected by structural drivers and $|F - Fs|$% represents the percentage of live mutants corresponding to faults undetected by state drivers. $|Ne|/|F - Fc|$ and $|Ne|/|F - Fs|$ represent the ratio of equivalent mutants out of live mutants undetected by structural drivers and by state

drivers respectively. For instance, the equivalent mutants correspond to 11.5% and 62% of undetected faults by state drivers in Carleton 1 for `Cruise Control` and `OrdSet`, respectively.

The number of live mutants is much higher than the number of equivalent mutants in the tested clusters. This suggests that a heuristic—previously used on other empirical studies on software testing [7, 10]—considering all faults undetected by any driver as equivalent mutants and eliminating them from the total set of mutants, cannot be applied in our case, especially in the case of `Cruise Control`. However, we can adjust, based on our qualitative analysis, the mutation scores of all the test drivers of our experiment. Let M_b and M_a be the mutation score of a driver before and after removing equivalent mutants respectively, N_e the number of equivalent mutants and F the total number of mutants. Then M_a can be computed as follows: $M_a = M_b * F / (F - N_e)$

We computed M_a for all drivers and clusters. Table 41 reports on the average mutation score of collected drivers per treatment ($|Fs|$% and $|Fc|$%) across experiments and clusters as well as their combinations average mutation score ($|Fs \ U \ Fc|$%) with and without considering the equivalent mutants. We tried to repeat the analyses of Chapter 4 after removing the equivalent mutants and, although mutation scores obviously increased, these changes turned out not to make any difference in terms of the conclusions we have drawn in the previous sections. A detailed, qualitative analysis of live mutants will be presented in Section 5.1.

			Mean of drivers mutation scores		
			state drivers	structural drivers	state + structural drivers
Carleton 1	Cruise Control	With equivalent mutants	24.47	27.69	35.86
		Without equivalent mutants	26.87	30.74	39.21
	OrdSet	With equivalent mutants	50.27	56.15	71.41
		Without equivalent mutants	55.81	62.34	79.57
Sannio 1	Cruise Control	With equivalent mutants	35.65	44.66	56.54
		Without equivalent mutants	39.02	48.88	61.82
	OrdSet	With equivalent mutants	71.76	70.31	83.66
		Without equivalent mutants	79.89	78.07	92.89
Carleton 2	Cruise Control	With equivalent mutants	30.8	50.13	57.13
		Without equivalent mutants	33.72	54.87	62.52
	Elevator	With equivalent mutants	35.45	40.54	51.33
		Without equivalent mutants	39.51	45.19	57.22
Sannio 2	Cruise Control	With equivalent mutants	34.55	46.89	55.32
		Without equivalent mutants	37.82	51.32	60.55
	Elevator	With equivalent mutants	35.06	36.97	48.26
		Without equivalent mutants	39.08	41.21	53.79

Table 41: Equivalent mutants' impact on mutation scores

4.6 Survey data analysis

After running the experiment, the participants were asked to answer and return a survey questionnaire. A number of questions were common to all questionnaires; the rest varied depending on the test technique used and on the lab order (Appendix L).

Each questionnaire is divided into three sets of questions: comprehension and lab time availability for the tasks implemented, test cases and oracles identification with the test technique used and the work environment. For each set of questions, we compare the answers collected across a number of factors: (1) cluster under test, (2) lab order, (3) participant ability, and (4) experiment. Results are presented in Appendix M and summarized below.

a. Regarding comprehension questions:

Participants agreed that:

- Having the model helped understanding the cluster under test. This observation was independent from the cluster tested, the experiment or the student ability.
- Understanding the cluster under test can be improved by providing the model.
- For all clusters, experiments, and abilities, lab instructions were clear.
- `Cruise Control` was easier to understand than `Elevator`. The latter was considered to be complex.

b. Regarding time availability questions:

Participants agreed that:

- For all clusters, experiments, and abilities, there were not enough time to complete the tasks. This is significantly more important in the case of `OrdSet` and `Elevator`.
- For `Elevator`, when writing test cases based on the state machine, it took too long to understand the cluster and its model. While for `OrdSet` and `Cruise Control`, most participants spent less than 50% of the lab time on the comprehension tasks, for `Elevator`, most participants spent more than 50% on the same tasks. Among those, 23% spent more than 75% of the lab time on the comprehension of the cluster and its model.

c. Regarding test cases and oracle identification questions:

Participants agreed that:

- State machine and transition tree (test technique) helped determine test cases. This observation is independent from the cluster tested, the experiment or the student ability.

- More information on the cluster under test is required to identify test cases and oracles (across all experiments).
- For `Elevator`, it is easier to identify test cases with state testing than with structural testing.

However, they were less supportive of the statement that state invariants and contracts help identifying oracles.

Participants were not sure (to slightly disagree) that equivalent test cases can be identified without basing test cases on the state machine.

*d. **Regarding work environment questions:***

Participants were knowledgeable of the Eclipse environment. They have often used it before the course. The other tool, the instrumented code, was found helpful for test cases identification.

Chapter 5

QUALITATIVE ANALYSIS

Results in Chapter 4 indicated a number of unkilled mutants higher for `Cruise Control` and `Elevator` and lower for `OrdSet`. To better understand why certain faults are difficult to detect by state drivers, we report on a qualitative analysis to identify what execution conditions would be required to detect those faults and whether these conditions were likely to be fulfilled by state testing or structural testing (Sections 5.1 to 5.3). Also, we report on a qualitative analysis (Section 5.4) aimed at better understanding the higher cost of state drivers compared to structural drivers. Finally, (Section 5.5) we investigate reasons for the limited structural coverage achieved by state drivers.

5.1 Qualitative analysis of live mutants

The main motivation for the qualitative analysis we performed on live mutants is to identify ways to improve the state testing strategy to increase its fault detection effectiveness. Another motivation is to classify faults detected by one technique and not by the other. To this aim, we identified the following disjoint sets of faults: (1) $F - (Fs \cup Fc)$, the set of all faults not detected by any driver, (2) $Fc - Fs$, the set of all faults detected only by structural drivers, (3) $Fs - Fc$, the

set of all faults detected only by state drivers, and (4) $Fs \cup Fc$, the set of all faults detected by both types of drivers. Table 42 reports, for each mutation operator, the total number of seeded mutants (|F|), the number of equivalent mutants (|Eq|) and the number of undetected mutants in each experiment. Then, we describe the procedure we followed to identify reasons for not detecting faults (Section 5.2). This procedure allowed us to classify undetected faults (Section 5.3).

Mutation operator	Cruise Control						OrdSet				Elevator			
	|F|	|Eq|	|F – Fs|				|F|	|Eq|	|F – Fs|		|F|	|Eq|	|F – Fs|	
			Car. 1	San. 1	Car. 2	San. 2			Car. 1	San. 1			Car. 2	San. 2
JTI											5	1	3	3
JTD											4	1	2	2
JSI	11	0	11	11	11	11	5	0	2	2	22	1	16	15
JSD	9	9	9	9	9	9	3	0	3	3	2	2	2	2
JID	9	9	9	9	9	9					6	3	3	3
JDC	1	0	0	0	0	0					1	0	0	0
EAM	5	0	5	5	5	5	4	1	1	1	56	9	43	39
AORB	28	0	28	28	28	27	90	0	6	0	76	0	55	53
AORS							8	0	0	0	6	0	0	0
AOIU	31	4	23	18	18	18	48	0	0	0	88	4	24	20
AOIS	148	5	106	73	79	69	297	55	77	53	556	76	99	90
AODU							4	0	0	0	5	0	5	4
ROR	79	6	56	44	47	46	47	6	10	7	107	11	48	45
COR							6	0	0	0	28	1	9	9
COD							1	0	0	0				
COI							4	0	0	0	28	0	4	5
LOI	49	0	24	20	21	21	107	0	4	2	192	12	61	58
ASRS	12	0	12	4	4	4					4	0	1	1

Table 42: Count of live mutants undetected by state drivers

5.2 Procedure to identify the reasons for not detecting faults

To identify the reasons for not detecting faults with a focus on faults that were not found at all $F - (Fs \cup Fc)$, and those found only by one technique and not the other ($Fs - Fc$ and $Fc - Fs$), we execute the corresponding mutants and generate execution traces. If the fault does not affect the output, the trace can then help us identify the reason by looking at intermediate values of local

variables and attributes: (1) to understand the reason why the fault did not propagate to the final output, and (2) to identify the limitation of the used oracle to detect such fault. Also, if the fault does indeed affect the output, the trace then helps us understand why the oracle did not detect any failure. In some cases, the oracle misses to check an attribute that has been affected by the fault.

An example of such a fault is one created by seeding a fault in the method `resizeArray()` of the `OrdSet` class as shown below (the index `k` highlighted in the code was replaced by `k++`):

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

Method `resizeArray()` is called whenever an element is to be added to a full set and that element is not already in the set. A resize can occur if two conditions are true: (1) the resized set size does not exceed the maximum set size (a constant), and (2) the number of resizes done on the set does not exceed the maximum resizes allowed (a constant). The fault get executed if those conditions are met, and when this is the case, the set is resized as expected, but with wrong content. An example of a test case that causes a failure if this fault is executed is to create an ordered set with content {1, 2, 3, 6}, and then add element "4" to the set. The result one gets is {0, 2, 0, 4, 6} instead of {1, 2, 3, 4, 6}. When executing the faulty program and generating the execution trace, we note that the resulting content of the set is faulty. In the original experiment,

Carleton 1, oracles included only state invariants assertions. No check of set content were done in those assertions. To detect the fault, it is necessary to check the exact content of the set. This can be done by verifying the class invariant or the `resizeArray` postcondition in the oracle. Such fault was not detected by Carleton 1's state drivers, while it was detected by Sannio 1's state drivers, where oracle helpers contained implementation of contract assertions.

As a result of this process, live mutants were divided into two sets: (1) the set of equivalent mutants, and (2) the set of remaining live mutants, which should be considered as undetected faults, and which are classified into categories and discussed in Section 5.3. Note that equivalent mutants represent 19% of live mutants in `Cruise Control`, 84% in `OrdSet` and 37% in `Elevator`. They are omitted from the remaining discussions as they do not represent faults.

5.3 Classification of undetected faults

This analysis was first performed on drivers collected from Carleton 1, with the aim of identifying changes to the experiment design and context for replications (Sannio 1, ...), as described in Section 3.4. The qualitative analysis on the collected drivers from the replications aimed at identifying further ways to improve the state machine testing strategy.

When combining the two test techniques, most faults seeded in `OrdSet` (mutants) were detected. Most remaining undetected faults do not affect the results of the faulty method and therefore correspond to equivalent mutants. For `Cruise Control`, JSI and JSD mutants (inserting and deleting a `static` keyword) were not killed as drivers created only one instance of the car. However, high numbers of other types of mutants were killed and the remaining live mutants were mostly due to the difficulty of devising precise oracles with exact values for class attributes. For `Elevator`, many undetected faults cause errors in the algorithms managing the behavior of

the cluster such as choosing the best elevator for a job, or the time it takes to move from a floor to another. `Elevator`'s state machine models only the algorithm managing the movements of an elevator. The other algorithms such as the selection of best elevator are not modeled in the state machine. Recall that the state machine represents only one elevator in the cluster and does not model the concurrent behavior of the cluster. Note that, in `Cruise Control` like in `OrdSet`, replication experiments' drivers killed more mutants mainly of type AOIS, ROR and LOI (Table 42). The qualitative analysis investigates in part the reasons for detecting those faults, and the factors in the replication experiments' design that impacted fault detection improvement.

As an outcome of the qualitative analysis, the identified categories of undetected faults are:

a. *Faults requiring precise oracles in order to be detected*: introducing such faults in the code causes changes in one or more attributes' computation that cannot be detected without precise oracles. This is the most frequent category in `Cruise Control` (Table 44). By using contract assertions in replications (Sannio 1, Carleton 2, Sannio 2), the number of undetected faults in this category dropped from the original experiment (Carleton 1), although a significant percentage of these faults (79%) remained undetected. To detect these faults, it would have been necessary to build an oracle that would nearly replicate the behavior of the code under test, which is not realistic in practice. The real-time behavior of `Cruise Control` was tested only by participants writing structural test cases. The availability of source code helped them to understand the algorithm managing the relationship between time and the value of class attributes such as speed and distance. In `OrdSet`, the use of contract assertions in oracles decreased the number of undetected faults belonging to this category from 32 (31% of undetected faults) in Carleton 1 to no fault in Sannio 1.

b. *Faults requiring specific scenario (e.g., a specific path in the state machine that is not covered by RTP) in order to be detected*, such as repeating some command call in `Cruise Control`. Specific sequences of events would be required in order for test case executions to reach specific attributes' values. Detecting some faults of this category would have required triggering an event (or a sequence of events) several times in the same test case (tested path). This, however, is not accounted for by the RTP testing technique, which limits the lengths of paths traversing the state machine graph, and therefore can be considered as a limitation of that technique. Some paths, even when they represent common usage scenarios of the system, are not necessarily covered by transition trees generated with the RTP testing technique. An example is the scenario of a real journey of an elevator: getting a number of requests, servicing them one after the other and finally stopping in the Idle state. This scenario, as well as many others, is not present in transition trees derived from the `Elevator` state machine. In `Cruise Control`, in order to be detected, some faults require calling a command (firing a transition) multiple times, as accelerating many times to get to the maximum throttle attribute value. Again, these scenarios are not part of transitions trees. In the replication experiments, sneak path testing was included as part of state testing (in addition to RTP). This decreased the undetected faults of this category for `Cruise Control`.

c. *Faults requiring specific parameter values to be passed in the test case* (such as specific set content in `OrdSet` or covering boundaries) *in order to be detected.* Such faults suggest that a combination of boundary analysis or category partition techniques with the state testing technique would be beneficial.

d. *Faults requiring specific execution time* (to allow for real-time attributes values to change*) in order to be detected.* These faults are specific to the real-time clusters. In `Elevator`, the number of undetected faults of this category was much lower than in `Cruise Control` because the real-time properties of `Elevator` are naturally modeled as state transitions and state actions in its state machine (Section 4.1.1.2). This category of faults is mainly observed in `Cruise Control` where state drivers did not execute for a long enough period of time (e.g., to reach the maximum speed) to detect faults in this category. This is the second important category of undetected faults in `Cruise Control` (25% of undetected faults – Table 44).

e. *Faults affecting static attributes*: modifying a class' static attribute into a non-static attribute, or vice versa. Those faults were not detected in any experiment. Test cases always included one instance of the class under test at a time. For instance, in `Elevator`, these faults were not detected by state drivers as the state machine models only one elevator.

f. *Faults causing wrong intermediate behavior without affecting the final outputs*: These faults are observed in `Elevator` when the exact behavior of each elevator is not checked by state drivers. For instance, an elevator may service a floor by going down first then going up instead of going up right away. The state machine does not distinguish between moving up or moving down, and how many floors are visited before the requested floor is serviced (as long as the requested floor is indeed serviced).

g. *Faults not observable based on the state machine*: these faults can be mainly observed in algorithms implemented in the source code, but not modeled by the state machines. This category represents the most important category in the `Elevator` cluster (51% of

undetected faults – Table 44). These faults are mainly observed in the algorithm managing the selection of the best elevator to service a job. This algorithm is not modeled by the state machine, and therefore state drivers could not detect the faults. In the `Cruise Control` cluster, such faults occur in the cruise control algorithm and in the algorithm implementing the real-time behavior of the simulated car.

h. *Faults affecting state behavior:* as a result of the execution of the fault, the state of the cluster is different from the expected state result. These faults went mainly undetected by structural drivers in `Elevator` where the state behavior is more complex than in the other two clusters. Implementing state invariants in oracle assertions of state drivers ensured the detection of those faults.

i. *Faults affecting class invariants*: as a result of the execution of the fault, the class invariant evaluates to a wrong value. A class invariant represents a set of constraints on class attributes values. Implementing class invariants in oracle assertions of state drivers ensured the detection of faults where an attribute's value is set out of specified boundaries. In `Elevator`, the number of elevators and floors defined in the group of elevators is specified in the class invariant. A fault changing such numbers is detected by state drivers. In `OrdSet`, the minimum and maximum size of an ordered set is specified as well in the class invariant. Having a set size smaller than the minimum size or greater than the maximum size is easily detected by state drivers.

j. *Faults affecting method results*: They may occur when the method post-condition depends on the object state. These faults might not be detected by structural drivers as they may require, to trigger failures, that the object is in a specific state.

Table 43 shows the distribution of undetected fault categories among the three subsets of undetected faults: (1) F – (Fs U Fc), the set of all faults not detected by any driver, (2) Fc – Fs, the set of all faults detected only by structural drivers, and (3) Fs – Fc, the set of all faults detected only by state drivers.

Category of undetected faults	F – (Fs U Fc)	Fc – Fs	Fs – Fc
Faults requiring precise oracles in order to be detected	x	x	
Faults requiring specific scenario	x	x	
Faults requiring specific parameters to be passed in the test case	x		
Faults requiring specific execution time	x	x	
Faults affecting static attributes	x	x	
Faults causing wrong intermediate behavior without affecting the final outputs	x	x	
Faults not observable based on the state machine	x	x	
Faults affecting state behavior			x
Faults affecting class invariants			x
Faults affecting method results			x

Table 43: Distribution of categories of undetected faults among testing techniques

The categories of faults detected by structural drivers and not detected by state drivers (Fc – Fs) appear to be a subset of the categories of faults undetected by any driver. In fact, while most of the faults belonging to these categories were undetected by any strategy, a number of them were, however, detected by structural drivers.

Table 44 reports, for each cluster, the counts of mutants corresponding to each category of undetected faults and what these counts represent as percentages of live mutants not including equivalent mutants. An empty cell means that there is no mutants in the corresponding category for the cluster under test. The results reported in Table 44 correspond to (cumulative) data collected from replicated experiments (i.e., Carleton 2, Sannio 1 and Sannio 2).

Category of undetected faults	Elevator	Cruise Control	OrdSet
Faults requiring precise oracles in order to be detected		65 (38%)	
Faults requiring specific scenario	18 (8%)	20 (12%)	
Faults requiring specific parameter values to be passed in the test case	8 (3%)		3 (25%)
Faults requiring specific execution time		43 (25%)	N/A
Faults affecting static attributes	14 (6%)	11 (6%)	2 (17%)
Faults causing wrong intermediate behavior without affecting the final outputs	35 (15%)	2 (1%)	
Faults not observable based on the state machine	116 (51%)	25 (15%)	1 (8%)
Faults affecting state behavior	6 (3%)	1 (1%)	
Faults affecting class invariants	7 (3%)	1 (1%)	6 (50%)
Faults affecting method results	25 (11%)	2 (1%)	

Table 44: Mutant count (percentage) per category of undetected faults and per cluster

5.4 Investigating the variation in cost

Results in Section 4.3 suggest that the cost—in terms of test case generation cost (LOC)—of structural testing is higher than that of state testing. Therefore, a qualitative investigation of the reasons for such an outcome is warranted.

The results of our qualitative analysis show that the main cause for cost variability in structural drivers is a high level of redundancy in test cases, where multiple test cases partially cover the same nodes and edges. Redundancy is limited for state testing as the test cases are precisely specified by a test strategy (RTP), leading to a limited redundancy when coding drivers. Another source of variability is the ineffective use of the available public methods to implement pieces of functionality required to create test drivers. For example, in `OrdSet`, a set can be created with two constructors, one creates an empty set and another creates a set with content from an array of integers. Some participants did not use the second constructor to create a non-empty set. Instead, they created an empty set and iteratively added elements to it with the "add one element" method. This un-necessarily increased the number of called methods in their drivers considerably. One factor causing variability in state machine drivers' cost is the number of

implemented RTPs—varying widely especially for `OrdSet` (from 10% to 100%) and for `Cruise Control`, where, in order to cover sneak paths, the number of covered RTPs for experiment replications was 25, including 13 sneak paths, instead of the 12 of Carleton 1.

Another factor causing variability in state machine drivers' cost is the variation in precision of implemented oracles (state invariant assertions only, state invariant + class invariant assertions, or state invariant + contract assertions). When profiling the execution of a driver, one can identify methods that are called the most. In the case of state testing, getters were the most called. This indicates that oracle precision heavily contributes to the high cost of state drivers. This is confirmed by the results obtained in the replications where cost increased significantly from the first experiment after increasing oracle precision by including contract assertions. Profiling results showed that the number of calls to getters in the replications was much higher than in the first experiment. In Section 7.2 we list some suggestions on how to improve state drivers' cost by lowering oracles' cost.

We can summarize the factors that contribute to a high cost of state drivers as follows:

a. The use of precise oracles after every event call[11] (include high number of method calls).

b. A number of RTPs have common sub-paths (i.e. initial setting may be common to a number of RTPs and repeated in a number of test cases), and therefore common code coverage.

c. A high number of RTPs.

d. A poor usage of public methods mainly in test case setup.

---

[11] This is confirmed by the profiler used to calculate the number of method calls. A high number of getters were called by state machine drivers. Recall that getters are used to get attributes values and therefore they are used extensively in oracle checks. These oracles are more expensive in Sannio 1 than in Carleton 1 as they include contract assertions in addition to state invariant assertions.

5.5 Analysis of code coverage

Another point worth being qualitatively investigated is the code coverage achieved by state drivers. In our experiments, uncovered nodes and edges are due to:

a. ***Methods not triggered by the state machine tests nor by oracles*** (state invariants + contracts): Some of the public methods are not triggered by state machine events, actions or activities, nor by oracles (state invariants and contracts checks), either directly or indirectly. These methods do not model functionality or behavior, but they implement helper methods(e.g., `toString()` methods) or getters.

b. ***Untested functionality***: Some cluster functionalities are modeled in the state machine but not exercised, or at least not fully exercised from the state drivers. For instance, in `Cruise Control`, the "Cruising" functionality, i.e. keeping the speed of the car at a fixed value, is modeled in the state machine but not fully exercised by the state drivers. This mainly depends on the real-time characteristics of the `Cruise Control` cluster. In `Elevator`, the state machine does not distinguish between moving the elevator up or down as only one state "moving" models both behaviors. The state drivers covered that state, however only letting the elevator moving up because the elevator is intitialized to start its operation from the ground floor, and therefore moving would always start as moving up. The constraints of the RTP testing technique on the generation of paths in the state machine would not allow the generation of a path in which the elevator would move up to get to some floor, and then change direction and move down.

c. ***Catching exceptions***: A number of the exception handling nodes and edges were not exercised by state drivers. Only two nodes and edges in this category are found in `Elevator`. Testing strategies dealing with exception coverage are proposed in [44, 87].

d. ***Unmodeled functionality***: The state machine does not model all cluster functionalities. This is the case of `Elevator` where the concurrent behavior of several elevators is not modeled by the state machine, which models only one elevator instance.

e. ***Handling boundaries and unexpected entries***: A number of nodes and edges handling boundaries and unexpected entries have not been exercised in state drivers. For instance, the code handling a non recognized command in `Cruise Control` was not exercised in state drivers.

For each tested cluster, Table 45 shows the categories to which correspond the uncovered nodes and edges. These results correspond to data collected from all experiments. An empty cell means the corresponding category does not apply to the cluster under test.

	OrdSet	Cruise Control	Elevator
Unaccessed methods by state machine and oracles	x		x
Untested functionality		x	x
Catching exceptions			x
Unmodeled functionality			x
Handling boundaries and unexpected entries		x	

Table 45: Classification of uncovered code

Chapter 6

THREATS TO VALIDITY

This chapter discusses the main threats to validity [93] that can affect our experiments.

6.1 Conclusion validity

Threats to conclusion validity are concerned with issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome of the experiment [93]. In our experiments, threats to conclusion validity could be essentially due to low statistical power, resulting from the relatively small number of participants. We were limited by the number of students enrolled in the testing courses within which we conducted the experiments. To limit the impact of this threat on our conclusions, we designed the experiment in such a way that each group would work on a different treatment for two subsequent labs, and thus doubled the number of observations. Replicating the experiment four times also increased our capability to identify significant results. In addition, we performed a power analysis (Section 4.1.1.1 – c) with the aim of determining the minimum effect size corresponding to the 80% power.

Statistical conclusions were supported by proper tests, in particular t-tests for pairwise comparisons and ANOVA for the analysis of co-factors. As discussed in Section 4.1.1, results of

t-test were also confirmed by an equivalent, non-parametric test (Wilcoxon). To account for possible Type I error chance capitalization, due to repeated significance testing, the significance level of the different t-tests was adjusted using Holm's procedure [48] which is less conservative than Bonferroni adjustment.

6.2 Internal validity

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. One example of internal validity threats is the learning and fatigue effects that can occur when the experiment is run more than once with the same participant groups. This threat is addressed in our experiment by using different treatments and different class clusters in each of the labs conducted for each group, and doing so in an order that prevents fatigue or learning effects to be confounded with our treatment.

To tackle the selection threat that is related to the variation in human performance, we identified a number of blocks to which the students were assigned. These blocks are based on marks achieved in earlier courses on software engineering and design. Students were selected from the different blocks to obtain a stratified random sampling over the different groups [93].

Another internal validity threat, the diffusion or imitation of treatments, was also limited by monitoring the labs and preventing the access to the experiment material outside the lab hours and by other groups' members. Note that the experiment material was accessed through the

course website only during lab hours with an address only known during the lab by members of the group working in that specific lab[12].

As opposed to participants working with structural testing, those working with state machines were not instructed to use equivalence class partitioning or boundary analysis to identify test cases parameters values. This was not necessary as in this experiment no boundary analysis was needed: the `Cruise Control`'s state machine has no parameters, and boundary analysis of the `OrdSet` and the `Elevator` clusters was already accounted for when exercising the guard conditions.

The allocated time for the experiments was fixed and limited: this sometimes caused a limited code and state machine coverage. This is usually unavoidable in the context of controlled experiments in artificial settings. We address this issue in the data analysis by using coverage (state machine and code) as an interaction factor. From a more general standpoint, it is often the case that, in controlled experiments, experimenters should choose between assessing the impact of a treatment on either the time to perform the tasks or their effectiveness within specific time constraints, but it is usually impractical to address both [11]. The work presented here matches the latter case.

Although participants were aware of the lab objectives, they did not know exactly what hypotheses were tested. Evaluation apprehension is also avoided, since participants were told that their labs were marked based on the capability of correctly following the lab procedure (and thus also correctly applying the testing strategy), rather than on the performance of the test drivers they produced.

---

[12] This was only applied during Carleton 1 and 2. For Sannio 1, the material was available only during lab time on a specific address known by participants. For Sannio 2, some setting problems prevented similar solution for material distribution, therefore the material was distributed to participants during lab time.

Another threat to internal validity is the use of the original, correct source code to generate their test cases based on code coverage analysis rather than using the faulty code on which the test cases are executed. The reason for this approximation is that it is impractical, if not impossible, to provide faulty code to subjects: the number of mutants varies from 382 to 1176 in the three clusters, and this would have required generating different test cases for each mutant. Also, providing mutated code to participants would have allowed them to discover where the mutants were seeded by simply comparing the different files, and this would have been a major threat. It should be kept in mind, however, that the differences between each individual mutant program and the original program are very small.

6.3 Construct validity

This type of threat is mainly related to our use of mutation analysis to measure the fault-detection effectiveness of testing strategies, as (i) the types of faults seeded may not be representative of "real" faults and (ii) there might be faults not produced by the chosen mutation strategy [64]. However, the existing literature [7, 8, 35] suggests that faults seeded using mutation operators can be representative of real faults. Since so far no results contradicting the above studies have been reported, relying on mutation to compare test techniques is practical as it provides large, automatically generated fault samples. The fact that we used three class clusters with very different code characteristics—thus leading to very different samples of mutants—should, however, limit the likelihood of this threat.

6.4 External validity

External validity relates to the external aspects that interact with the treatments and limit the generalization of the results. The selection of fourth-year engineering students as participants in

Carleton 1 and 2 and Masters' students with little industry experience in Sannio 1 and 2 could be a threat to external validity as they may not be representative of the population of professional software developers. However, it is well-known that productivity can vary a level of magnitude between the best and worst developers. Students at Carleton were overall good Java developers, as this is the main language used throughout their four years of study. Moreover, they are better acquainted with UML, and in particular state machine modeling, than most average practitioners: they passed two full term courses on the subject. Students in Sannio already hold an engineering graduate degree, were also trained in Java programming, and were carefully selected based on their academic track record. In addition, they all went through thorough testing training prior to the experiments. So, overall, for the specific tasks at hand, the participants in our experiments can be considered competent. Issues related to the use of students in experiments are discussed in previous literature [9, 50, 52], which indicates that, often, advanced students and professionals are statistically similar in various performance measures. In conclusion, the existing literature suggests that, though our results might not generalize to all experienced, professional developers, existing evidence suggests they are probably representative of junior and intermediate developers. However, in some cases the work pressure in industry—e.g., in the context of a major project releases—is different from that of an academic lab. To alleviate this problem, we used strictly-enforced, fixed laboratory time.

The particular choice of the class clusters to test in this experiment may be considered an external validity threat: results can always be somehow specific to the software under test. Moreover, for controlled experiments, class clusters' size must be limited to allow enough time for performing the tasks in laboratory, controlled settings. However, while the class clusters we used could appear relatively simple and small compared to large, industrial systems or

subsystems, we believe they still can be considered representative of some of the situations where state testing would be applied. Though they differ in terms of complexity, the `Cruise Control` and the `Elevator` clusters are representative of real-time, reactive classes with a state-dependent behavior, where class attributes are evaluated based on elapsed time between events and the current cluster state. For instance, the current floor attribute for an elevator, which specifies the floor number the elevator is currently on, depends on its current state and on the time elapsed after an event such as "a request for the elevator to go up". The `OrdSet` class, on the other hand, is modeled by a large state machine with complex guard conditions. It is representative of classes encapsulating complex data structures with large transition trees (30 round-trip paths). These two common categories of class clusters are usually modeled using state machines in UML-based development [27, 46, 57]. This is further supported by industrial case studies reported in the literature [30, 49]. Additionally, it is very uncommon in practice to model subsystems or entire systems using state machines, as this is far too complex in realistic cases. To assess testing techniques on such large programs, one would have to resort to industrial case studies. Controlled experiments involving humans in artificial settings necessarily use smaller programs but they have, however, the advantage of achieving high internal validity. They must, therefore, be often complemented with industrial case studies that are usually strong on external validity (realistic, possibly larger systems, time constraints, etc.) but weak in terms of avoiding confounding factors.

The state machines for Cruise Control and Elevator do not fully model the two clusters under test. They only model the state behavior of the clusters. This is often the case in practice. State machines are rarely used to completely capture the system behavior, as this would result in too large and unmanageable models. In conclusion, state testing results, in practice, can significantly

depend on the level of abstraction of the models used, in other words of the abstraction gap between the model and its implementation. This is particularly significant when modeling large programs, where abstraction becomes a necessity due to the complexity of the source code under test. It is therefore possible that our results would be significantly different on much larger programs and their corresponding models, which would be at a higher level of abstraction than that of the models used in our experiments. One plausible effect would be a decrease in fault detection effectiveness of state testing and an even greater necessity to complement it with structural testing. In general, choosing an appropriate level of detail when developing state machines is an important decision, that may be in part driven by testing objectives [90].

Chapter 7

PROPOSITIONS OF IMPROVEMENTS TO STATE TESTING

Based on the qualitative analysis from Chapter 5, we propose a set of improvements to the state testing strategy to increase its fault-detection effectiveness (Section 7.1) and lower its cost (Section 7.2).

7.1 Propositions for fault-detection effectiveness improvement

Unmodeled behavior in state machine is one issue that limits fault detection effectiveness of the state test technique. The following summarizes the main causes for the limited behavior modeling in state machines:

1. ***Not every behavior is state related***: For instance the variation in class attributes values of `Cruise Control` over time when the car is running is not state related. The "Running" state as originally modeled for this cluster does not depend on the variation of speed, distance or throttle class attributes. Those class attributes are managed by an algorithm which can possibly be better modeled with an activity diagram. Another example is the concurrency in `Elevator` which is not state related. Each state in `Elevator`'s state machine represents

only one `Elevator` instance and therefore cannot model the concurrency between several `Elevator` instances.

2. ***Impractical modeling***: When modeling with a state machine, the designer may decide that it is impractical or too complex to model some behavior in the state machine. Again, the real-time behavior of `Cruise Control` falls in this category. Modeling it in the state machine would lead to an impractical model as in Figure 34, 0.

3. ***Abstract vs. concrete states***: When modeling with a state machine, one of the decisions to be taken by the designer is the level of abstraction of states. The more abstract the states are, the fewer low level attribute variations appear in the state machine.

4. ***Taking away implicit transitions***: Not every event lead to state changes. It is common to omit implicit transitions that do not affect current state and have no output to simplify the model.

The qualitative analysis of live mutants in Carleton 1 helped us identify a number of additions to our state testing strategy in order to improve the fault detection ratio of state drivers [70] (RQ8). We applied a number of those additions to the replications and the analysis of their outcome helped us identify additional issues and suggest more improvements. The following is a list of all proposed improvements:

1. The use of activity diagrams to model algorithms with high control flow complexity such as the activity diagram in Figure 29 modeling a running car, or the selection algorithm for best elevator provided in Figure 30. These diagrams would help ensure minimal edge coverage for the most complex methods. The coverage criterion would be to cover all paths in the activity diagram. If there is a loop in the activity diagram, then typical loop coverage heuristics

should be applied [16]. In the case of real-time clusters, these algorithms are often driven by timers and model changes as a function of time.

2. Increase oracle precision. This can be achieved by including contract assertions (class invariant and methods' pre and post conditions) in oracles in addition to state invariants.
3. Complement the round-trip path technique with sneak path testing. This has been recommended by Binder [16] and our data provides empirical evidence that this is required.
4. In the case of real-time clusters, strengthening oracle assertions with Boolean expressions that reflect the time behavior of the cluster. These expressions define how cluster attributes or class attributes are expected to change over time. For instance, in `Cruise Control`, the throttle of the car is supposed to decay over time because of air resistance. It is expected that such oracles would be hard to implement as they likely require the implementation of complex algorithms.
5. High-stress[13] and most common[14] use case scenarios. Often, paths in the state machine correspond to high-stress or common use case scenarios, but the RTP technique does not cover them because of constraints on the generation of the transition tree: For instance, a path cannot include the same transition more than once. A use case description would help to identify steps of the scenario that correspond to a path in the state machine. A combination with a black-box technique for parameter selection, such as category-partition or boundary analysis would be beneficial to better specify the test case (e.g. test case parameters, number of transitions to fire in a loop ...). As an example, in `Cruise Control`, testing the boundaries of speeding and braking is a critical test case. In order to implement such a test case, accelerating more than once by following the transition "accelerate" on the Running

[13] We define a high-stress use case scenario as a test sequence aimed at stressing the system under test in order to make it peform its tasks at difficult levels.

[14] A most common use case scenario corresponds to a scenario that often occurs in real situations.

state is required. Note that no path in the transition tree includes those repeated transitions (accelerate on Running state).

6. Combining boundary analysis with round-trip path technique to test for boundaries. A RTP may be implemented more than once in different test cases and with different parameter settings.
7. Including oracles for intermediate outputs. For instance, in `Elevator`, one may verify the floors traversed to service a request. The direction of traversal and the number of traversed floors are to be checked as intermediate outputs.
8. Some seeded faults can only be detected if the execution outputs are precisely compared with expected outputs. Such precise oracles are, however, in practice very expensive to implement, especially when concurrency results in a certain level of indeterminism in the produced ouputs. Because of its excessive cost and the fact that the above seven recommendations appear to yield satisfactory results, this last suggestion is not investigated in the remainder of the section.

An example for recommendation 1 is the activity diagram for a running car in `Cruise Control` depicted in Figure 29. Such an activity diagram helps to identify test cases that cannot be otherwise identified based on the state machine and using the round-trip path technique. For instance, to perform boundary testing in the case of `Cruise Control`, we need to test the case where a car is running at maximum speed. However, to reach the maximum speed, the car should accelerate for some time as speed is a function of time (it changes every 200 ms). The same transition on the Running state would be traversed a number of times in the same path to implement such a test case. This will not correspond to any round-trip path as specified by the RTP criterion. This test case corresponds to executing the highlighted path in Figure 29 a number

of times. Therefore, test driver execution time should be extended to allow speed to increase. Recall that the time-related behavior was not modeled in `Cruise Control`'s state machine.

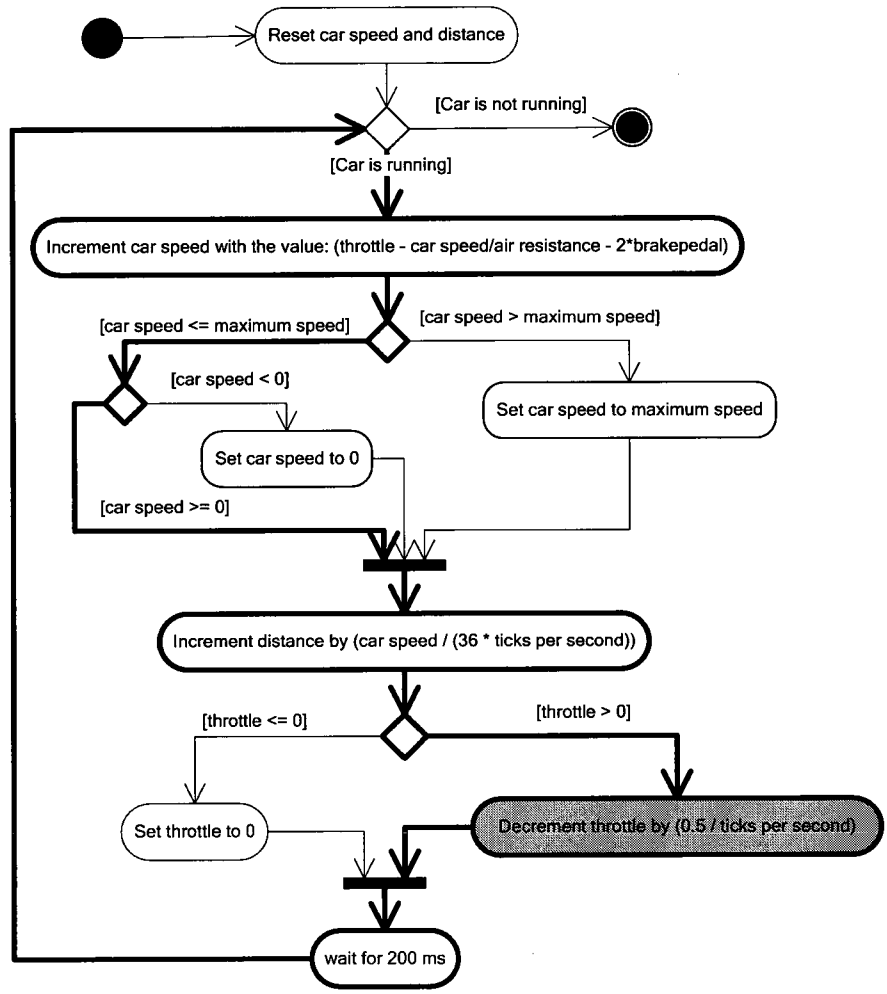


Figure 29: Cruise Control - Running car activity diagram

Note from the activity diagram that the speed is a function of the car throttle. The throttle value is increased by a fixed amount for every accelerate command (refer to method "accelerate" postcondition in Appendix K.1.3) and it decays with time: activity highlighted in grey. Therefore, in order to continue to bring speed to its maximum value, the command "accelerate" should be repeated a number of times. Covering all edges in the activity diagram implies that some edges would be traversed a number of times before being able to cover other edges. For example, to

cover the path in the activity diagram where a speed is greater than maximum speed, the car speed should first reach maximum speed. But in order to reach the maximum speed the highlighted path in Figure 29 should be executed a number of times.

The activity diagram also models the impact of time on the different attributes of the cluster. Every 200 ms, the speed, throttle, and distance are updated. Including such behavior in the state machine would lead to impractical and highly complex state machines (Figure 34 in Appendix A). The actions on the added transitions are of complex computational nature, as complex as the source code they are modeling. Fully covering them would require careful analysis of the different conditional statements and identification of the test parameters. As discussed before, in practice, modeling a state machine would be limited to state-dependent behavior. Implicit transitions that do not affect current state and have no output are usually taken away from the state machine to reduce complexity. In the case of `Cruise Control`, timer controlled transitions fall in this category of implicit transitions and therefore are omitted from the state machine to simplify it.

The activity diagram in Figure 30 describes the algorithm for optimal elevator selection. An instance of the elevator cluster may include more than one elevator. When multiple requests are sent to the cluster, this algorithm identifies the best elevator to service each request. `Elevator`'s state machine does not model concurrency in the cluster as it models the behavior of only one elevator instance. Covering paths in the activity diagram (Figure 30) is therefore necessary to address concurrency issues.

In a model-based testing strategy, activity diagram-based testing can complement state testing in different ways. As described above, a state machine cannot realistically model all relevant behavior. For example, real-time behavior is sometimes hard to model in a state machine as it

leads to impractical and complex state machines (Figure 34 in Appendix A). This would in turn lead to a significant cost increase when applying the RTP test technique or other state test techniques. Another important use of activity diagrams is to model algorithms with high control flow complexity (first improvement recommendation in the list above). Critical and common use case scenarios (suggestion 5) can also be represented in activity diagrams. An example was provided above addressing testing `Cruise Control` at maximum speed. Boundary analysis can also be used when covering activity diagrams (suggestion 6), as paths in the activity diagram can be specified by ranges of attributes values. As exemplified above, testing the boundaries of these ranges is also necessary in the context of state testing.

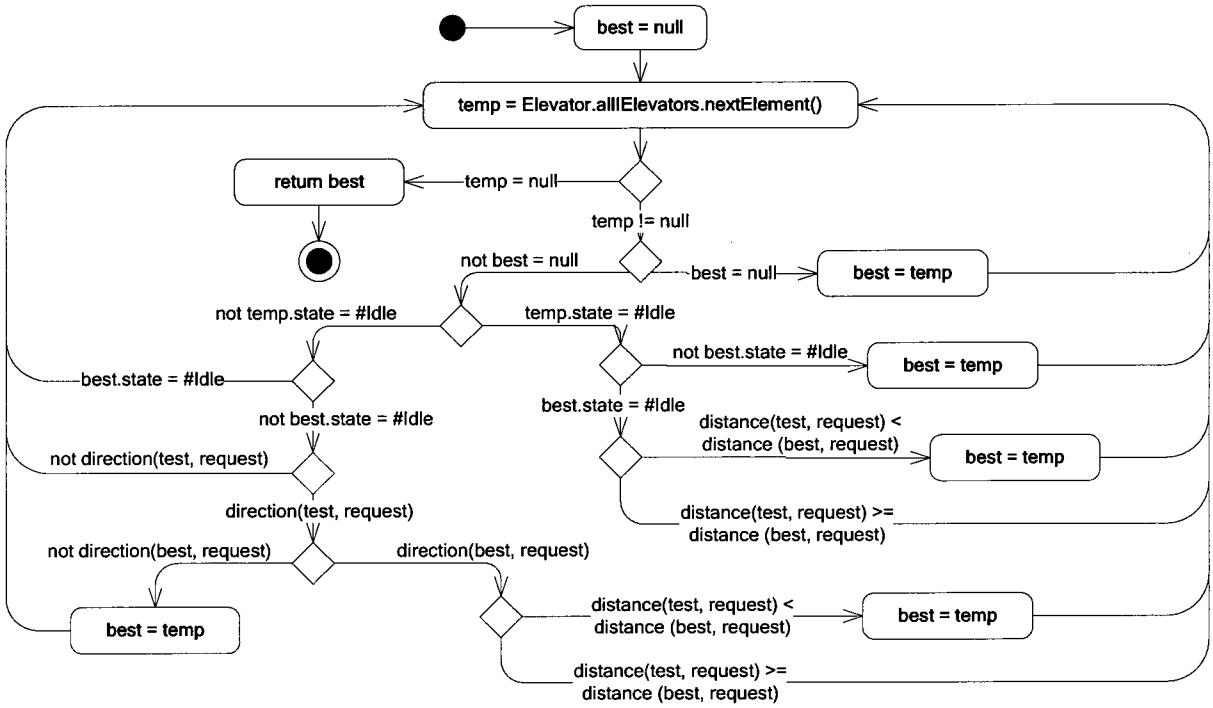


Figure 30: Elevator's activity diagram for best elevator selection algorithm

In order to assess the impact of the proposed changes on the fault detection effectiveness of state drivers, we applied suggested changes to one complete driver of each class cluster. The results are presented in Table 46, Table 47 and Table 48 below. These tables present the following

values: 1) mutation scores of drivers when implementing one or more changes; note that equivalent mutants were not accounted for, (2) the increase in mutation score achieved when compared to both the initial driver (not including any change) and the driver implementing only state invariants in oracles, (3) node and edge coverage showing the impact on code coverage when implementing changes to drivers. The relative impact on fault detection effectiveness varies much from one suggestion to another. For `Cruise Control`, the suggestion with the highest impact on fault detection effectiveness was implementing a high-stress use case scenario. Note that only one such use case scenario was implemented as an example. For `Cruise Control`, the high-stress use case scenario implemented was to speed up to reach maximum speed then breaking to the maximum to reach speed zero. The `Cruise Control` cluster was stressed to make it work up to its limits.

	Mutation score	Increase in mutation score	Node Coverage	Edge Coverage
RTPs + state invariants	26.65	-	78.16	62.05
+ activity diagram coverage	44.70	18.05	86.18	76.60
+ sneak paths coverage	32.95	6.30	85.37	76.78
+ contract assertions in oracles	36.39	9.74	79.83	64.53
+ contract assertions & sneak paths coverage	40.68	14.03	82.58	74.96
+ contract assertions & activity diagram coverage	49.86	23.21	82.45	73.62
+ sneak paths & activity diagram coverage	50.14	23.49	88.07	82.24
+ contract assertions & sneak paths coverage & Strenghtening oracle assertions[15]	43.27	16.62	84.46	74.96
+ contract assertions & sneak paths coverage & activity diagram coverage	55.01	28.36	88.07	82.24
+ contract assertions & sneak paths coverage & activity diagram coverage & Strenghtening oracle assertions	55.59	28.94	94.51	88.85
+ high-stress use case scenario[16]	51.01	24.36	88.07	76.6
+ common use case scenario[17]	44.42	17.77	92.7	79.57
+ boundary analysis[18]	37.25	10.60	87.17	74.78
+ oracle for intermediate output[19]	43.84	17.19	85.37	71.14
+ all listed implemented changes	70.82	44.17	97.21	94.3

Table 46: Impact of proposed changes to state testing strategy on fault detection effectiveness of Cruise Control driver

The next two suggestions with the highest impact for `Cruise Control` were activity diagram coverage and implementing common use case scenarios. These two suggestions have a common characteristic: they require the execution to last long enough to allow speed to increase to a certain level. This implies better coverage of the code and consequently better mutation scores. Recall that `Cruise Control` state machine did not include any reference to time constraints and therefore when covering the state machine, a driver misses an important part of the code that relates to real-time behavior.

---

[15] Oracles had been strengthened by testing speed, throttle and distance for positive values.
[16] Implementing high-stress use case scenario: Speeding up to maximum speed then braking to the maximum to get to speed zero.
[17] Implementing common use case scenario: Speeding for some time, then cruising for some time, then braking to stop and turn off the engine.
[18] Implementing tests for boundaries of the maximum throttle and maximum brake values that can be achieved.
[19] Implementing oracle for testing distance value overtime. A distance should be increasing over time after accelerating.

	Mutation score	Increase in mutation score	Node Coverage	Edge Coverage
RTPs + state invariants	87.89	-	94.59	90.48
+ boundary analysis[20]	89.50	1.61	94.59	90.48
+ contract assertions in oracles	95.38	7.49	95.49	92.06
+ contract assertions & Strenghtening oracle assertions[21]	96.80	8.91	95.49	92.06
+ contract assertions & Strenghtening oracle assertions & boundary analysis	96.80	8.91	95.49	92.06

Table 47: Impact of proposed changes to state testing strategy on fault detection effectiveness of OrdSet driver

For `OrdSet`, whose state machine describes the cluster behavior in a comprehensive and precise manner, implementing the proposed changes did not significantly increase the mutation score (a total of 8%): Table 47. Recall from the analysis of live mutants in Section 5.1 that most of the remaining mutants are equivalent mutants. The highest increase in mutation score was provided by including contract assertions in oracles (started in Sannio 1).

`Elevator`, the most complex among the three studied clusters, provides many challenges for testing. Its real-time behavior and concurrency, and the complexity of its code and state machine, as well as the limited lab time, resulted into unsatisfactory fault detection levels by state drivers. When applying the proposed changes to a complete state driver, we observed an improvement in fault detection ratios. The combined changes led to an increase in mutation score of 22.5% to a total of 76%. If we eliminate equivalent mutants from the calculation of the mutation score, we achieve 98.65%, which can be considered highly satisfactory. Looking at Table 48, one can notice that the changes with the highest impact on mutation score were testing common and high-stress use case scenarios, and implementing oracles for intermediate outputs. The latter can be

[20] Applying boundary analysis to set size.
[21] Oracles had been strengthened by testing last value index.

attributed to the complex nature of the cluster in which the execution of an event leads to a large number of actions and activities.

	Mutation score	Increase in mutation score	Node Coverage	Edge Coverage
RTPs + state invariants	59.62	--	73.24	55.4
+ activity diagram coverage	65.88	6.26	79.88	60.41
+ sneak paths coverage	66.91	7.29	76.49	60.34
+ contract assertions in oracles	66.35	6.73	74.9	57.73
+ contract assertions & sneak paths coverage	72.04	12.42	78.15	62.68
+ contract assertions & activity diagram coverage	70.99	11.37	83.19	65.68
+ sneak paths & activity diagram coverage	70.99	11.37	83.51	68.24
+ contract assertions & sneak paths coverage & Strenghtening oracle assertions[22]	75.64	16.02	78.15	62.68
+ contract assertions & sneak paths coverage & activity diagram coverage	76.57	16.95	85.2	70.16
+ contract assertions & sneak paths coverage & activity diagram coverage & Strenghtening oracle assertions	76.94	17.32	85.2	70.16
+ critical use case scenario + common use case scenario[23]	67.67	8.05	82.71	64.13
+ boundary analysis[24]	65.97	6.35	76.11	60.44
+ oracle for intermediate output[25]	67.86	8.24	73.65	56.33
+ all listed implemented changes	84.74	25.12	86	71.04

Table 48: Impact of proposed changes to state testing strategy on fault detection effectiveness of Elevator driver

Combining all the propositions with the basic RTP technique implementation in a driver causes a very high increase in mutation score (a 44% increase in `Cruise Control`). Code coverage increase was also important. Ninety seven percent of node coverage was achieved in `Cruise Control`, 95% in `OrdSet` and 86% in `Elevator`. We hypothesize that implementing more use case scenarios to address other common or critical behaviors would increase the mutation score further.

[22] Oracles had been strengthened by verifying direction of elevator movement.
[23] (1) requests for concurrent elevators, (2) multiple requests for one elevator, and (3) requests in different directions.
[24] Testing floor requests boundaries.
[25] Implementing oracle for testing the distance value overtime. A distance should be increasing over time after accelerating.

7.2 Suggestions for cost improvements

Results discussed in Section 5.4 showed that state drivers were more expensive than structural testing drivers when considering the LOC metric or the number of method calls metric to measure drivers' cost, especially so in the replications where more precise oracles were used based on contract assertions. This can significantly limit the practical use of this approach to testing in industrial contexts.

When performing a qualitative analysis of test drivers, specific factors were identified as heavily contributing to the cost of state drivers. The main factors affecting such cost were: (1) high number of RTPs, (2) common subpaths in RTPs, and (3) common boolean expressions in contract and state invariant assertions. Based on this qualitative analysis we propose the following list of improvements to lower the cost of the state testing strategy:

1. Omitting repeated oracle assertions in test cases implementing common sub-paths. This suggestion addresses main factor '2'.

   Let S be the set of states: $S=\{s_i,\ 1\leq i\leq n\}$ where n is the total number of states, E the set of events: $E=\{e_j,\ 1\leq j\leq m\}$ where m is the total number of events, and G the set of boolean expressions representing the guard conditions: $G=\{g_k,\ 1\leq k\leq w\}$ where w is the total number of guards.

   A path in the state machine can be denoted as follows: $P = s_i\ (\rightarrow e_j[g_k]^{?}\rightarrow s_t)^{*}$ [26] where $1\leq i\leq n$, $1\leq j\leq m$, $1\leq k\leq w$, and $1\leq t\leq n$.

[26] In a regular grammar, the notation "?" is used to indicate zero or one occurrence of an expression, notation "*" is used to indicate zero, one or more occurrences of an expression, and notation "+" indicates one or more occurrences.

A common subpath is the occurrence in two different paths P_1 and P_2 of a common chain "$(\rightarrow\ e_j[g_k]\rightarrow s_t)+$" [26]. Common sub-paths can occur at any location in paths, at the beginning of the paths (i.e., starting after the initial state), in the middle of the paths, or at the end of the paths. A sub-path to two paths P_1 and P_2 can occur at different locations in P_1 and P_2: For instance, P_1 may start with a subpath sP which may occur at the end of P_2. More than one common subpath can exist between two paths and a common subpath can exist between two or more paths. Recall that RTPs, or round trip paths, are simply paths in the transition tree and are derived using a breadth or depth first search traversal of the graph representing the state machine, thereby generating paths that have parts in common.

The following is a simple example, let $RTP_1 = state_1 \rightarrow event_1 \rightarrow state_2 \rightarrow event_2 \rightarrow state_3$ and $RTP_2 = state_1 \rightarrow event_1 \rightarrow state_2 \rightarrow event_3 \rightarrow state_4$ (guard conditions have been omitted for illustration purposes), if test case T_1 implements RTP_1 and test case T_2 implements RTP_2, and T_2 is executed after T_1, then in test case T_2 we can omit the oracle check after $event_1$ as the transition from $state_1$ to $state_2$ after triggering $event_1$ is tested in T_1.

2. Making changes to the transition tree generation procedure of the RTP test technique. This suggestion addresses main factor '1'.
   a. Using of depth-first traversal instead of breadth-first traversal. A depth-first traversal would generate longer but fewer RTPs, therefore less setup and initialization code would be required in test cases. Recall that the two approaches are supposed to be equivalent as they both cover round trip paths.
   b. Changing the stopping criterion in the transition tree algorithm: i.e., using transition instead of state. Recall that in the original RTP technique [16], a tree node is considered to be terminal when the state it represents is already present anywhere in the tree or is the final

state. Alternatively, the stopping criterion could focus on transitions instead of states. A path in the transition tree, generated by the depth-first traversal of the state-machine, ends when no new transition (i.e. a transition that is not present anywhere in the tree) can be added to the end of the path. The traversal of the state-machine continues with other paths until all transitions are covered by the tree. An exception to this rule allows the traversal of a transition more than once in the case where a subsequent transition can be only traversed after a repeated traversal of a previous transition. By changing the stopping criteria, we aim at reducing the number of repeated transitions in the resulting transition tree (one of the limits discussed previously). This would lead to a decreased number of method calls. For instance, when applying the original RTP technique to `Cruise Control`'s state machine, transition `Inactive/Idle -> engineOn -> Active` appeared in 19 paths / test cases (Figure 36). However, when applying the suggested stopping criteria, this transition appeared only in two paths / test cases (Figure 37).

3. Simplifying oracle checks by simplifying boolean expressions. This suggestion addresses main factor '3'. Recall that we have suggested the use of contract assertions in addition to state invariant assertions in oracles. Contracts and state invariants may have parts in common, thus leading to redundant checks, and therefore oracles may be simplified by putting them in disjunctive form. For instance, a class invariant may be simply a collection of the different state invariants. In that case, there is no need to include class invariant in the oracle check, and simply check the state invariant. An example is the class invariant of class `Controller` from `Cruise Control`, that includes the state invariants of the four possible states of the class, namely Inactive, Active, Cruising, and Standby:

```
context Controller
inv: (self.controlState = INACTIVE) implies
       (self.sc.state=DISABLED and self.sc.setSpeed >= 0
```

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

4. Wise use of public methods in a way to minimize the number of method calls. For instance, in `OrdSet`, a non-empty set can be created in one of two ways:

   i. Using a constructor with an array parameter: `OrdSet s = new OrdSet({1,2,3});`

   ii. Creating an empty set then adding elements, one by one, with the method "`add`":

   ```
   OrdSet s = new OrdSet();
   s.add(1); s.add(2); s.add(3);
   ```

   In the first example, creating the set required one method call, however in the second, it required four method calls.

5. Eliminate initialization code assertions. For instance, if an `OrdSet` test case requires the creation of a non-empty set at initialization, there is no need to verify if the created set is not empty. For instance, in the example below, there is no need for testing if the newly created set is empty. Including state invariants in oracle assertions makes sure this check is done in oracles without need to add initialization code assertions.

```
OrdSet s = new OrdSet();
if (!s.isStateEmpty()) {
    System.out.println("Test case failed");
    return;
}
```

6. Use of `setState()` methods to bypass already tested subpaths in RTPs. Following the state pattern would ensure the implementation of `setState()` methods in context classes, when this is practical and feasible.

To assess the impact of these suggestions on state drivers' cost reduction, we applied these improvements to a number of participant drivers. Note that the proposed improvements do not apply to all drivers and cannot always be combined. Therefore, we randomly selected a number of drivers (18 in total) from the entire collection of drivers from all experiments and all clusters, and applied the proposed changes to them. The results are presented below in Table 49. The table reports on the change in driver's cost measured with the number of method calls metric and on the change in mutation score. Recall that state testing was found to be less effective than structural testing when considering this metric (Section 4.3). Only one change is applied to each driver each time. The numbers presented in the table for the "applied change" correspond to those of the list of proposed improvements above.

Driver	Experiment	Cluster	Applied change	Original cost	Cost after change	Cost decrease (%)	Original mutation score	Mutation score after change
1	Carleton 1	Cruise Control	"1"	486	368	24.28	24.87	23.56
2	Sannio 1	Cruise Control	"2"	849	316	62.78	40.8	39.27
3	Carleton 2	Cruise Control	"3"	1539	952	38.14	39.79	39.79
4	Sannio 2	Cruise Control	"1"	1165	763	34.51	39.53	39.53
5	Carleton 1	Cruise Control	"5"	497	437	12.07	24.09	23.3
6	Sannio 1	Cruise Control	"6"	1521	882	42.01	38.22	36.65
7	Carleton 2	Cruise Control	"1"	1541	893	42.05	39.53	37.17
8	Sannio 2	Cruise Control	"2"	1541	512	66.77	39.53	37.43
9	Sannio 1	OrdSet	"1"	2490	2366	4.98	85.9	85.9
10	Sannio 1	OrdSet	"2"	2490	2059	17.31	85.9	85.9
11	Sannio 1	OrdSet	"3"	1415	1374	2.9	80.13	80.13
12	Carleton 1	OrdSet	"4"	561	539	3.92	38.46	37.82
13	Carleton 1	OrdSet	"5"	561	551	1.78	38.46	36.22
14	Carleton 1	OrdSet	"6"	2136	1992	6.74	79.17	78.69
15	Carleton 2	Elevator	"1"	7907	7206	8.87	65.14	64.84
16	Carleton 2	Elevator	"2"	7907	2046	74.12	65.14	63.79
17	Sannio 2	Elevator	"3"	1059	1059	0	32.91	32.65
18	Sannio 2	Elevator	"1"	2457	2318	5.66	51.02	51.02

Table 49: Application of changes to improve drivers cost as number of method calls

The difference between the original cost of drivers (measured as number of method calls) and the modified cost varies considerably from a driver to another and from a specific change to another. The second proposed improvement, namely the use of depth-first traversal transition trees, showed the best impovements in terms of lowering drivers' cost (highlighted in gray in Table 49). We can explain this as the second proposed improvement is the one that reduces most the number of test cases to implement. In `Cruise Control`, the number of RTPs to implement dropped from 25 with breadth-first traversal to 3 with depth-first traversal, which corresponds to more than 60% decrease in terms of number of method calls (Table 49). The reduction in RTPs was also considerable in `Elevator` were it dropped from 40 RTPs to 7 RTPs.

The reduction was more conservative in `OrdSet` where the number of RTPs dropped from 30 to 22. This explains the higher cost reduction in `Elevator` and `Cruise Control` (grayed lines in Table 49), corresponding to drivers 2, 8 and 16, when compared to the reduction for `OrdSet` (driver 10 in Table 49). To the difference with the other two clusters, the number of transitions going out of the initial state is large (20 transitions for `OrdSet` compared to one transition in `Elevator` and `Cruise Control`). This corresponds to 20 different paths in the transition tree for both depth-first and breadth-first searches. Another difference between `OrdSet` and `Cruise Control` is that `OrdSest`'s state machine has guard conditions on transitions which is not the case for `Cruise Control`. Guard conditions limit the number of possible sequence in the transition tree.

The suggestions with the lowest cost reduction ratios were suggestions 3, 4 and 5 from the list above. They try to eliminate some unnecessary testing code without lowering the number of test cases (or number of calls in test cases). They address specific cases that exists only in few

drivers. Therefore, their impact on cost reduction is limited and depends on the extent to which the problems they address occur in the drivers, such as testing initialization code.

The difference in mutation score between the original drivers and the changed ones after applying all proposed improvements show no significant drop in mutation scores, that varied in the 1-2 % range. This drop in mutation scores can be explained by the fact that a number of the suggested changes to the testing technique imply a reduction of oracle assertions in test drivers. For instance, for the first suggestion discussed in this section, oracle assertions are omitted from common sub-paths in test drivers. A path is basically a test case specification. When creating a test case from a path one has to identify test inputs. Two different test cases may have a common subpath, i.e., sequence of states and transitions, but they may use different input values for the events in the common subpath. Different inputs may exercise slightly different parts of the code. Not checking oracles in all occurences of common sub paths may therefore reduce fault detection.

Recall that the objective for the suggested changes in this section is to reduce state testing cost. When defining testing strategy, one may make compromises between fault detection effectiveness and testing cost.

To have a complete picture of the impact of the suggestions for cost improvement, we calculated the LOC and CPU time for those drivers, before and after the application of the changes. The results are presented in Table 50. Similarly to the results of number of method calls, the highest decrease in LOC was for the second proposition for improvement and for the same drivers. The change applied to drivers lowered the number of implemented RTPs and consequently lowered LOC considerably. For `OrdSet`, this suggestion had not the highest impact on LOC as the

number of implemented RTPs was not lowered much (22 RTPs instead of 30, compared to 3 RTPs instead of 25 in `Cruise Control`, and 7 RTPs instead of 40 in `Elevator`).

Driver	Experiment	Cluster	Applied change	Original LOC	LOC after change	LOC decrease (%)	Original CPU time	CPU time after change	CPU time decrease (%)
1	Carleton 1	Cruise Control	"1"	247	209	15.38	15	15	0
2	Sannio 1	Cruise Control	"2"	633	228	63.98	15	16	6.25
3	Carleton 2	Cruise Control	"3"	748	678	9.36	15	16	6.25
4	Sannio 2	Cruise Control	"1"	775	544	29.81	15	16	6.25
5	Carleton 1	Cruise Control	"5"	368	345	6.25	15	15	0
6	Sannio 1	Cruise Control	"6"	759	484	36.23	15	16	6.25
7	Carleton 2	Cruise Control	"1"	604	413	31.62	16	16	0
8	Sannio 2	Cruise Control	"2"	604	230	61.92	16	16	0
9	Sannio 1	OrdSet	"1"	948	818	13.71	16	16	0
10	Sannio 1	OrdSet	"2"	948	791	16.56	16	16	0
11	Sannio 1	OrdSet	"3"	640	638	0.31	16	16	0
12	Carleton 1	OrdSet	"4"	571	538	5.78	16	16	0
13	Carleton 1	OrdSet	"5"	327	260	20.49	16	16	0
14	Carleton 1	OrdSet	"6"	687	601	12.52	15	15	0
15	Carleton 2	Elevator	"1"	1956	1321	32.46	98031	98031	0
16	Carleton 2	Elevator	"2"	1956	719	63.24	20109	98031	79.49
17	Sannio 2	Elevator	"3"	700	694	0.86	10219	10235	0.16
18	Sannio 2	Elevator	"1"	1049	800	23.74	71844	72141	0.41

Table 50: Impact of the application of propositions on LOC and CPU time of selected drivers

For CPU time, the changes did not make any noticeable impact to the cost of `Cruise Control` drivers. Recall that `Cruise Control` state machine did not address the real-time behavior and that their execution was almost instantaneous. No impact was found on CPU execution time for `OrdSet` drivers as well. For `Elevator`, only one driver showed an exception and its cost measured in CPU time was lowered considerably (79%). The change applied was again the

second suggestion, which led to a considerable decrease in implementd RTPs (seven, down from 40) and consequently to eliminate the repetition of many paths in the state machine that contribute to the total execution time.

Chapter 8

LESSONS LEARNED

This chapter reflects on the lessons learned from executing a series of experiments. During both the design and the execution of the experiments we encountered a number of issues that we detail in the next paragraphs and explain how we addressed them. Reflecting back on the learned lessons would help us identify some practical guidelines that can be useful for researchers in the field who may encounter similar issues.

First, we address design issues which include: lab time, clusters to test, and context selection. From the experience we gained in this study we hypothesize that lab time is an important factor in experiment design that can deeply affect the results. Increasing lab time would allow participants to better understand the clusters under test as well as the tasks to be executed and would probably allow them to complete more tasks. However, most of the times, lab time can not be indefinite. Therefore the choice of the length of the lab time should be based on a rough measure of the required time by the average participant to complete the tasks. We suggest that the experimenter asks a third party, i.e., a colleague or a trainee, to execute the tasks in the

experiment environment to have an approximate measure of lab time. This measure can be adjusted to the closest possible lab time based on other constraints such as lab availability.

Selecting the clusters to test is very important for the design of experiments. Two related variables are particularly important: (1) the size of the cluster, and (2) its complexity. It is recommended to choose relatively small clusters from a variety of domains to avoid the threat of limited time on the conclusions deduced. Also, the selected clusters should be of limited complexity to limit the likelihood that the complexity of the cluster would affect the ability to draw the correct conclusions about relations between the treatment and the outcome of the experiment.

To the difference with professional participants, when dealing with students it is often hard to convince them to work seriously on the experiment tasks unless they are rewarded. It is therefore recommended when working with students as participants to reward them on their work. This would motivate them to complete the tasks and follow the instructions more closely.

Next we address preparation issues. Experimentation requires a lot of preparation, i.e. documentation, code samples, lab setup, etc. This may lead to a number of mistakes that should be avoided: (1) missing document, (2) wrong versioning, (3) wrong participant assignment, and (4) data loss. For instance, the experimenter may forget to upload a document to the server used during the experiment leading to time lost while recovering the missing document. Another mistake that can occur is to provide a wrong version of the document losing a number of the updates that have been included. Also, a participant may be mistakenly assigned to the wrong group, as a result of this mistake, his/her work may not be considered for data analysis. All these mistakes can be avoided by using configuration tools and available experimentation frameworks.

Also, the literature provides a number of useful insights from the learned lessons from experiments that can be of great help for new experimenters [11, 20, 29, 56, 85].

Documentation is an important tool in experimentation. Both documentation completeness and clarity contribute to a successful experimentation. We recommend to experimenters, while preparing for the experiment, to ask a third-party (someone who has not worked on the preparation of the experiment, neither is an identified subject) to evaluate the clarity and completeness of the instructions and the documents included in the experiment package. Also we recommend providing simple examples in the documentation that can be useful for the participants to avoid misunderstanding of task execution and to avoid spending too much time on understanding instructions.

Another issue to discuss that is related to the preparation of the experiment is the knowledge level of the techniques used in the experiments. Prior to the experiment, participants should have full understanding of the techniques. Being unsure of how to use a technique, or having little experience with the tools leads to lost lab time to understand the "how to"s. It is therefore recommended to let the participants exercise on similar tasks beforehand. For instance, in the case of student participants, assignments or labs prior to the experiment can be used to ensure a sufficient level of knowledge of the tools and techniques to be used in the experiment. The experimenter can participate in the preparation and the execution of those assignments and labs to make sure participants are well trained. This may also help highlight some issues in the experiment plan such as unclear documentation.

When reflecting on execution issues, one difficult and hard issue to address is the possible communication of data and documents across participants, groups and labs. To limit these types of information communication, experimenters should provide lab setup in which the

communication between participants is limited. For instance, participants from different groups would be seated in the lab in alternating rows to limit discussions within group. Proctors would make sure no discussion or passing information (directly or on the network) is undergoing during the experiment. Also, documents should be accessed only by participants needing them for the tasks and only during lab time.

The discussed issues so far apply in general to any type of software experimentation. In addition to those general issues, we encountered some specific issues related to the assessment and comparison of test techniques. One main issue is the choice between real faults and seeded faults. While with real faults the results may be considered more realistic, it is usually difficult to find systems to test with enough known faults with various types. Seeded faults based on mutation operators address those issues by generating large numbers of mutants with different types and locations. However, mutation leads to another issue which is equivalent mutants. Some methods are proposed in literature to detect equivalent mutants but their detection rate is around 50% only. The most common practice is to manually identify equivalent mutants, but this is time-consuming and error-prone. We suggest that researchers and experimenters in the software testing field share a pool of systems and their faults to minimize the work associated with preparing the systems and their fault sets. The infrastructure proposed in [39] is one such place for sharing resources.

Fault detection rate is the main metric of comparison of test techniques. Therefore, checking the outcome of a test case is of particular importance. However, based on the output of the test case, it is not always straightforward to answer the question: "is the test case successful?" Specifying oracles with Boolean output is not always possible, or even practical. For instance, it is hard to verify textual outputs especially when the order of outputs is undeterministic. We encountered

this problem in the case of `Elevator` where a number of elevators may be running concurrently and each of them would be writing messages to the output. When verifying if a test case is successful or not, one may encounter another issue which is the occurrence of a deadlock or an infinite loop: A test case may not terminate. It is up to the experimenter to decide whether deadlocks and infinite loops should be considered as wrong behavior and consequently consider the test cases causing them to be unsuccessful. Threads monitoring the execution of drivers can detect such wrong behavior when a driver's execution time exceeds its expected execution time.

Running a large number of drivers manually is a hard task especially when running drivers on mutants. Scripts automating the execution of drivers would considerably simplify the task. In addition to the execution of drivers, the scripts can be used to collect data about fault detection. Reports on detected and undetected faults would be generated and used afterwards in a qualitative analysis of the collected data.

The steps required for the application of a test technique may not all be specific to that technique. Those steps that are general may or may not be common between the test techniques assessed and compared during the experimentation, but they are usually time-consuming. On the other hand, laboratory time allocated for experimentation is often limited. Therefore, to limit the time spent during the experimentation on tasks specific to the assessed techniques, the experiment designer is encouraged to provide participants with resources simplifying the general tasks. For instance, categories of parameters may be provided based on category-partition to simplify test cases' parameter selection task. Control-flow graphs of tested methods, transition trees of state machines are other examples of resources that can be communicated to participants to simplify some of the experimentation tasks.

Another issue that may be encountered with experimentation on test techniques is having unclear or coarse definitions of test techniques. To make sure that all participants apply the assessed test techniques in the same way and as expected from the experimenters, clear definitions and descriptions of the test techniques should be provided. Ambiguities in the description of the application of a test technique may lead to different interpretations by different participants.

Chapter 9

GUIDELINES ON THE USE OF STATE TESTING

Considering the results of our empirical study, the qualitative analysis presented in Chapter 5, we provide some practical guidelines on the use of state testing. The goal is to improve fault detection effectiveness and code coverage of state testing, while limiting its cost. Chapter 7 presents empirical evidence on the effectiveness of what is suggested in these guidelines to improve state-based testing strategies.

a. **State testing is not a silver bullet as it cannot deal with all systems' properties.** A state machine is not able to represent all the properties of a system that would be relevant for implementing it, and therefore also for thoroughly testing it. In particular, it is often not practical to model all real-time and concurrency properties (Section 4.1.1.2). Real-time properties may be difficult to model in a state machine especially when other factors affect attributes' variations over time (e.g., car speed is determined through a relationship involving air resistance, throttle, and current speed over time). Also, concurrency adds considerable complexity to state models. These limitations of state testing were outlined by the results of the qualitative analyis (Section 5.3 – items b, d, f, and g and Section 5.5 – items b and d).

Separating concerns, i.e., state-dependent behavior vs. time dependent behavior or concurrent behavior, addresses this complexity by taking modeling of real-time and concurrency properties away from the state machine and modeling them with other means. Therefore, we consider that state testing is sufficient by itself only when the state machine represents precisely and completely the behavior of a sequential component with no real-time and concurrent behavior. However, for complex real-time or concurrent components, a model-based testing strategy should not be limited to state testing. A complete model-based testing strategy would include other model-based testing techniques such as a use case-based testing technique to cover common use case scenarios and high-stress use case scenarios not covered by RTPs. Techniques based on activity diagrams or sequence diagrams would be required to fully exercise methods with complex control flow or functionality representing the cluster's real-time behavior.

b. **Sneak-paths should be covered by state testing.** For the sake of simplification, self-transitions with no actions are often omitted when modeling state machines. Qualitative analysis (Section 5.3 – item b) and the results of the replication experiments (Section 4.1.1.1) proved the importance of including sneak path testing to improve state testing fault detection effectiveness. Therefore it is recommended to complement the original RTP technique with sneak-path testing as it was originally suggested by Binder [16].

c. **Other functional testing technique(s) to complement state testing.** The complexity of state machines varies considerably from one cluster to another. One of its factors is the number of parameters in events and actions. Although an RTP can be implemented in more than one test case with different parameter values, the RTP technique coverage criteria is fulfilled by simply implementing one test case per RTP. This suggests that the choice of parameter values

is of great importance to the fault detection effectiveness of the RTP technique. Hence our recommendation is to combinine the RTP technique with boundary analysis and/or category partition testing to address items c and d in Section 5.3 and item e in Section 5.5. The latter has been empirically validated and recommended in [20].

d. **One must reach an appropriate compromise between highly precise oracles and testing cost.** Increasing oracle precision proved to improve fault detection effectiveness but also increase the cost of state testing (Section 5.3 – item a and Section 5.4 – item a). Testers might want to reach a compromise between having a highly precise oracle, and keeping the testing cost down—in terms of resource consumption and time needed to run the test cases. For instance, we observed (see Section 5.4 – item b) that RTPs have common subpaths and that, therefore oracle checks for events (and actions) in these common subpaths are performed several times (each time the common sub-path appears in a test case). To reduce the testing cost, we recommend to only check oracles in common subpath once. The oracle cost can also be reduced by simplifying Boolean expressions in oracle checks, i.e., by putting them in a disjunctive form.

e. **Useful heuristics can be used to reduce the cost of state testing.** State testing cost can also be reduced by reducing the number of RTPs in the generated transition tree (see Section 5.4 – item c). To this aim, a depth-first traversal of the state machine would often generate less, but longer, RTPs, with a smaller number of method calls in total, than breadth-first traversal. Common subpaths in depth-first traversal RTPs are expected to be reduced as well since there are fewer paths. This is supported by the examples in Section 7.2.

f. **Complementing state-based testing with structural testing is cost-effective when state-machines do not fully capture the class cluster behavior.** As shown in Section 4.4, if state

machine models are very detailed—e.g., as it happens for `OrdSet`—the cost-effectiveness of complementing state-based drivers with structural drivers (ΔMs/ΔCs) is relatively low, as the added drivers are not able to exercise a substantial, additional portion of the source code (not covered yet by state drivers), and thus detect additional faults. These additional drivers might be expensive in terms of source code to be written and of methods to be invoked (as they require developing precise oracles), thus the overall cost-effectiveness of complementing state drivers with structural drivers is generally low. On the other hand, the cost-effectiveness of complementing state drivers with structural drivers is higher (measuring the cost in terms of method calls or drivers's LOC, see gray columns in Table 39) for state machine models which are not precise enough to fully capture the class cluster behavior. This is the case for `Cruise Control` and `Elevator`, where real-time behavior and exceptions are not captured by the state machine models.

Chapter 10

CONCLUSIONS

This thesis investigated, through a series of controlled experiments involving human participants (senior, carefully trained undergraduate students and experienced graduate students), the fault-detection effectiveness and cost of state testing of class clusters with state-driven behavior. This is of practical importance as state-driven testing has been often recommended for complex class clusters in the literature [18, 20, 25, 30, 31, 74, 76]. To provide a baseline of comparison we compared state testing with structural testing based on code coverage analysis, which can be considered a common, widely adopted practice for testers. Furthermore we investigated whether the two strategies are complementary in detecting faults and could be combined. Finally, we investigated factors that may affect the effectiveness of these testing strategies.

Results show—in a context where testers have limited time, and where state machines closely (but realistically) model the functionalities of the cluster—that testing driven by code coverage analysis is not less effective at detecting faults than a well-known strategy for state testing, i.e., the W-method [32] or round-trip path testing for UML state machines [16]. However, the two test strategies seem to be complementary in terms of faults they are able to detect. This suggests

that they should probably be used together, as opposed to being considered as alternatives. Since state machines are often produced before code, and since testing based on code coverage analysis is notoriously tedious and time consuming, it is probably wise to first test class clusters based on state machines and then complement test suites based on coverage analysis.

The obtained results also suggest that the effectiveness of state testing strongly depends on the nature of the software under test, the state machine model itself, but also the choice of test oracles. Our investigation shows that state testing alone is not sufficient to test class clusters with real-time and concurrent properties. Other model-based testing technique would be required to complement testing and to address areas and functionalities of the code not precisely modeled by the state machine. For example, in some cases, it may be advisable to complement the state machine with activity, timing or sequence diagrams describing properties not fully modeled by state machines (e.g., modeling the way time-dependent class attributes are updated), and then trying to cover such diagrams to complement state testing. Our results also show the benefits of using precise oracles based on contract assertions and class invariants, as well as the usefulness of testing illegal and implicit transitions in state machines.

The results of the replicated experiments confirm the results of the original experiment. Therefore, we believe that the results we provide in this thesis and the conclusions we draw provide useful insights to practitioners and researchers alike and confirm our beliefs of the importance of model-based testing in software testing.

To further complement the quantitative analysis, we performed a qualitative analysis to investigate the reasons for limited fault detection effectiveness and the high cost of state drivers. Having used only three class clusters for the experiments, it is of course difficult to determine the extent to which quantitative results can generalize. However, the qualitative analysis allowed us

to identify the main reasons why test techniques performed the way they did. And these reasons are in no way specific to the clusters under test and should therefore apply to many other class clusters.

Based on the analysis of live mutants, we proposed a set of changes to be applied to the RTP technique and to the state testing strategy in order to improve its fault detection effectiveness. Assessing those changes after applying them to complete state drivers of the three clusters under test shows important impact on improving state drivers' mutation scores.

When considering the generation and execution cost of the test techniques, state testing is found to be more costly than structural, code coverage testing. Oracles are found to be the main reason for this high cost as high oracle precision leads to high test driver execution cost. We presented in this thesis a set of improvements to be applied to the state testing technique with the aim of addressing the high generation cost of state drivers. A preliminary application of these improvements to a random set of drivers showed indeed their potential in lowering drivers' generation cost. No significant side effects of these changes were observed: Variations in mutation scores of the modified drivers were very low. This allows us to conclude that the proposed changes improve drivers' generation cost without reducing their fault detection effectiveness.

Preparing, conducting and managing a series of experiments was not an easy task. We learned many lessons from this experience the hard way. By sharing those lessons with other researchers and experimenters and combining them with other experiences, we believe that a common knowledge in software experimentation can be built.

Future work will investigate how state testing can be improved—even augmented with other models—to detect faults that have shown to be detected only by structural drivers and how its

cost can be decreased. The ultimate goal is to rely exclusively on model-based testing to avoid tedious source code coverage analysis and the manual derivation of test cases.

Replicating the experiment in an industrial setting is another future work that would address the external validity threats of using students as participants and using small clusters to test. By replicating the experiment with industrial applications and with professionals we would aim to further confirm the results of this study.

We provided in this study a number of suggestions to lower state testing cost. We applied them to few drivers. The observed results were promising. But the reduction of cost came with a decrease in mutation score. Future work will investigate reasons for the decrease in mutation scores and propose improvements to the cost reduction suggestions so that lowering cost would not affect fault detection effectiveness.

Using graph theory, we plan to study possible relationship between graph properties of state machines, such as connectivity, and drivers' cost when using depth-first search or breadth-first search to generate transition trees. This should possibly lead to yet other kinds of improvements to round trip path testing.

List of publications

1. S. Mouchawrab, L. C. Briand and Y. Labiche, "A measurement framework for object-oriented software testability", *Information and Software Technology*, vol. 47 (15), pp. 979-997, 2005.
2. S. Mouchawrab, L. C. Briand and Y. Labiche, "Assessing, Comparing, and Combining Statechart- based testing and Structural testing: An Experiment", *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, Madrid, Spain, pp. 41-50, 2007.

3. S. Mouchawrab, L. C. Briand, Y. Labiche and M. Di Penta, "Assessing, Comparing, and Combining State Machine-Based Testing and Structural Testing: A Series of Experiments", *Forthcoming in IEEE Transactions on Software Engineering,* 2010.

Chapter 11

REFERENCES

[1] Eclipse, http://www.eclipse.org, (Last accessed 2010)

[2] Eclipse Metrics plugin, http://sourceforge.net/projects/metrics, (Last accessed 2010)

[3] Eclipse Test and Performance Tools Platform project (TPTP), http://www.eclipse.org/tptp, (Last accessed 2010)

[4] UML 2.0 OCL Specification, http://www.omg.org/cgi-bin/doc?ptc/2003-10-14, (Last accessed 2010)

[5] Wikipedia, http://www.wikipedia.org, (Last accessed 2010)

[6] Software-artifact Infrastructure Repository (SIR), http://sir.unl.edu/portal/index.html, (Last accessed 2010)

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

[8] J. H. Andrews, L. C. Briand, Y. Labiche and A. S. Namin, "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria," *IEEE Transactions on Software Engineering*, vol. 32 (8), pp. 608 - 624, 2006.

[9] E. Arisholm and D. I. Sjøberg, "Evaluating the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software," *IEEE Transactions on Software Engineering*, vol. 30 (8), pp. 521-534, 2004.

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

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

[12] B. Beizer, *Software testing techniques*, Van Nostrand Reinhold Co., 2nd Edition, 1990.

[13] P. J. Bernhard, "A reduced test suite for protocol conformance testing," *ACM Transactions on Software Engineering and Methodology*, vol. 3 (3), pp. 201-220, 1994.

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

[15] R. Binder, "Design for Testability in Object-Oriented Systems," *Communications of the ACM*, vol. 37 (9), pp. 87-101, 1994.

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

[17] G. V. Bochmann and A. Petrenko, "Protocol testing: review of methods and relevance for software testing," in *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*. Seattle, Washington, United States: ACM, 1994.

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

[19] L. C. Briand, "A Critical Analysis of Empirical Research in Software Testing," *First International Symposium on Empirical Software Engineering and Measurement*, Madrid, Spain, 2007.

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

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

[22] L. C. Briand, Y. Labiche and J. Cui, "Automated Support for Deriving Test Requirements from UML Statecharts," *Special Issue of the Journal of Software and Systems Modeling (Springer)*, vol. 4 (4), pp. 399-423, 2005.

[23] L. C. Briand, Y. Labiche and Q. Lin, "Improving State-Based Coverage Criteria Using Data Flow Information," *Proceedings of the 16th IEEE International Conference on Software Reliability Engineering*, Chicago, USA, pp. 95-104, 2005.

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

[25] L. C. Briand, Y. Labiche and Y. Wang, "Using Simulation to Empirically Investigate state Coverage Criteria based on Statecharts," *Proceedings of the 26th International Conference on Software Engineering*, Edinburgh, Scotland, UK, pp. 86-95, 2004.

[26] A. Brooks, J. Daly, J. Miller, M. Roper and W. Murray, "Replication of experimental results in software engineering," Department of Computer Science, University of Strathclyde, Glasgow, Technical report ISERN-96-10, 1996.

[27] B. Bruegge and A. H. Dutoit, *Object-oriented Software Engineering: Using Uml, Patterns and Java*, Prentice Hall, 2004.

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

[29] J. Carver, L. Jaccheri, S. Morasca and F. Shull, "Issues in using students in empirical studies in software engineering education," *Proceedings of the 9th International Symposium on Software Metrics*, pp. 239-249, 2003.

[30] P. Chevalley and P. Thevenod-Fosse, "Automated generation of statistical test cases from UML state diagrams," *Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International*, pp. 205-214, 2001.

[31] P. Chevalley and P. Thevenod-Fosse, "An empirical evaluation of statistical testing designed from UML state diagrams: the flight guidance system case study," *Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development*, pp. 254-263, 2001.

[32] T. S. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Transactions on Software Engineering*, vol. SE-4 (3), pp. 178-187, 1978.

[33] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*, Lawrence Erlbaum Assoc Inc, Second Edition, 1988.

[34] J. Daly, A. Brooks, J. Miller, M. Roper and M. Wood, "Verification of results in software maintenance through external replication," *Proceedings of the International Conference on Software Maintenance*, pp. 50-57, 1994.

[35] M. Daran and P. Thévenod-Fosse, "Software error analysis: a real case study involving real faults and mutations," in *ACM SIGSOFT international Symposium on Software Testing and Analysis*. San Diego, California, United States, 1996.

[36] R. A. DeMillo, R. J. Lipton and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, vol. 11 (4), pp. 34-41, 1978.

[37] J. L. Devore and N. Farnum, *Applied Statistics for Engineers and Scientists*, Duxbury, 1999.

[38] A. Dmitrienko, G. Molenberghs, W. Offen and C. Chuang, *Analysis of Clinical Trials Using SAS: A Practical Guide*, SAS Publishing, 2005.

[39] H. Do, S. Elbaum and G. Rothermel, "Infrastructure support for controlled experimentation with software testing and regression testing techniques," *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pp. 60-70, 2004.

[40] T. Dybå, V. B. Kampenes and D. I. K. Sjøberg, "A systematic review of statistical power in software engineering experiments," *Information and Software Technology*, vol. 48 (8), pp. 745-755, 2006.

[41] G. A. Ferguson and Y. Takane, *Statistical Analysis in Psychology and Education*, McGraw-Hill Ryerson Limited, Sixth Edition, 2005.

[42] P. G. Frankl and S. N. Weiss, "An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria," *Proceedings of the Fourth Symposium on Software Testing, Analysis and Verification*, Victoria, British Columbia, Canada, pp. 154-164, 1991.

[43] P. G. Frankl, S. N. Weiss and C. Hu, "All-Uses versus Mutation Testing: An Experimental Comparison of Effectiveness," *Systems and Software*, vol. 38 (3), pp. 235-253, 1997.

[44] C. Fu, A. Milanova, B. G. Ryder and D. Wonnacott, "Robustness Testing of Java Server Applications," *IEEE Transactions on Software Engineering*, vol. 31 (4), pp. 292-311, 2005.

[45] E. Gamma, R. Helm, R. Johnson and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Computing Series, Addison-Wesley Professional, 1995.

[46] H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*, Addison-Wesley Professional, 2000.

[47] R. J. Harris, *A Primer of Multivariate Statistics*, Lawrence Erlbaum Associates, 2001.

[48] S. Holm, "A Simple Sequentially Rejective Multiple Test Procedure," *Scandinavian Journal of Statistics*, vol. 6, pp. 65-70, 1979.

[49] N. Holt, B. Anda, K. Asskildt, L. C. Briand, J. Endresen and S. Frøystein, "Experiences with precise state modeling in an industrial safety critical system," *Proceedings of Critical Systems Development Using Modeling Languages, Workshop in Conjunction with UML'06*, Genova, Italy, pp. 68-77, 2006.

[50] R. Holt, W., D. Boehm-Davis, A. and A. C. Shultz, "Mental representations of programs for student and professional programmers," *Empirical studies of programmers: second workshop*, Ablex Publishing Corp., pp. 33-46, 1987.

[51] H. S. Hong, Y. G. Kim, S. D. Cha, D. H. Bae and H. Ural, "A Test Sequence Selection Method for Statecharts," *Software Testing, Verification and Reliability*, vol. 10 (4), pp. 203-227, 2000.

[52] M. Höst, B. Regnell and C. Wohlin, "Using Students as Subjects - A Comparative Study ofStudents and Professionals in Lead-Time Impact Assessment," *Empirical Software Engineering*, vol. 5 (3), pp. 201-214, 2000.

[53] M. Hutchins, H. Foster, T. Goradia and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," *Proceedings of the 16th international conference on Software engineering*, Sorrento, Italy, 1994.

[54] P. Jorgensen, *Software Testing: A Craftman's Approach*, CRC Press, 2002.

[55] A. Kleppe, J. Warmer and W. Bast, *MDA Explained, The Model Driven Architecture: Practice and Promise*, 2003.

[56] O. Laitenberger and H. M. Dreyer, "Evaluating the usefulness and the ease of use of a Web-based inspection data collection tool," *Proceedings of the 5th International Software Metrics Symposium*, pp. 122-132, 1998.

[57] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* 2004.

[58] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines-a survey," *Proceedings of the IEEE*, vol. 84 (8), pp. 1090-1123, 1996.

[59] P. S. Levy and S. Lemeshow, *Sampling of Populations: Methods and Applications*, Wiley, 3rd Edition, 1999.

[60] C. M. Lott and D. Rombach, "Repeatable Software Engineering Experiments for Comparing Defect-Detection Techniques," *Empirical Software Engineering*, vol. 1 (3), pp. 241-277, 1996.

[61] G. Luo, G. v. Bochmann and A. Petrenko, "Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized Wp-Method," *IEEE Transactions on Software Engineering*, vol. 20 (2), pp. 149-162, 1994.

[62] Y.-S. Ma, Y.-R. Kwon and J. Offutt, "Inter-Class Mutation Operators for Java," *Proceedings of the 13th International Symposium on Software Reliability Engineering*, Annapolis, MD, 2002.

[63] Y.-S. Ma and J. Offutt, Description of Method-level Mutation Operators for Java, http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf, (Last accessed 2010)

[64] Y.-S. Ma, J. Offutt and Y. R. Kwon, "MuJava : An Automated Class Mutation System," *Software Testing, Verification and Reliability*, vol. 15 (2), pp. 97-133, 2005.

[65] B. Marick, *Craft of Software Testing: Subsystems Testing Including Object-Based and Object-Oriented Testing*, Prentice-Hall, 1985.

[66] D. E. Matthews and V. T. Farewell, *Using and Understanding Medical Statistics*, Karger Publishers, 2007.

[67] P. McMinn and M. Holcombe, "Evolutionary Testing Using an Extended Chaining Approach," *Evolutionary Computation*, vol. 14 (1), pp. 41-64, 2006.

[68] B. Meyer, "Applying Design By Contract," *Computer*, vol. 25 (10), pp. 40-51, 1992.

[69] J. Miller, "Applying meta-analytical procedures to software engineering experiments," *Systems and Software*, vol. 54, pp. 29-39, 2000.

[70] S. Mouchawrab, L. Briand and Y. Labiche, "Assessing, Comparing, and Combining Statechart- based testing and Structural testing: An Experiment," Carleton University SCE-06-15, 2006.

[71] K. R. Murphy and B. Myors, *Statistical Power Analysis: A Simple and General Model for Traditional and Modern Hypothesis Tests*, Lawrence Erlbaum, 1998.

[72] L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann and W. Grieskamp, "Optimal strategies for testing nondeterministic systems," *ACM SIGSOFT Software Engineering Notes*, vol. 29 (4), pp. 55-64, 2004.

[73] C. Nebut, F. Fleurey, Y. Le Traon and J.-M. Jezequel, "Automatic test generation: a use case driven approach," *IEEE Transactions on Software Engineering*, vol. 32 (3), pp. 140- 155, 2006.

[74] A. J. Offutt and A. Abdurazik, "Generating Tests from UML specifications," *Proceedings of the 2nd International Conference on the Unified Modeling Language*, pp. 416-429, 1999.

[75] A. J. Offutt and W. M. Craft, "Using compiler optimization techniques to detect equivalent mutants," *The Journal of Software Testing, Verification and Reliability*, vol. 4 (3), pp. 131-154, 1994.

[76] J. Offutt, S. Liu, A. Abdurazik and P. Ammann, "Generating Test Data From State-based Specifications," *The Journal of Software Testing, Verification and Reliability*, vol. 13 (1), pp. 25-53, 2003.

[77] J. Offutt and J. Pan, "Automatically Detecting Equivalent Mutants and Infeasible Paths," *The Journal of Software Testing, Verification and Reliability*, vol. 7 (3), pp. 165-192, 1997.

[78] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating fuctional tests," *Communications of the ACM*, vol. 31 (6), pp. 676-686, 1988.

[79] T. V. Perneger, "What's wrong with Bonferroni adjustments," *British Medical Journal*, vol. 316, pp. 1236-1238, 1998.

[80] A. Petrenko, N. Yevtushenko, A. Lebedev and A. Das, "Nondeterministic State Machines in Protocol Conformance Testing," in *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI*: North-Holland Publishing Co., 1994.

[81] P. Piwowarski, M. Ohba and J. Caruso, "Coverage Measurement Experience During Function Test," *Proceedings of the 15th International Conference on Software Engineering*, 1998.

[82] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch and T. Stauner, "One evaluation of model-based testing and its automation," *Proceedings of the International Conference on Software Engineering*, St. Louis, MO, USA, pp. 392 - 401, 2005.

[83] F. Shull, V. Basili, J. Carver, J. C. Maldonado, G. H. Travassos, M. Mendonca and S. Fabbri, "Replicating software engineering experiments: addressing the tacit knowledge problem," *International Symposium on Empirical Software Engineering*, pp. 7- 16, 2002.

[84] F. Shull, J. Carver and G. H. Travassos, "An empirical methodology for introducing software processes," *SIGSOFT Software Engineering Notes*, vol. 26 (5), pp. 288-296, 2001.

[85] F. Shull, J. Singer and D. I. K. Sjøberg, *Guide to Advanced Empirical Software Engineering*, Springer, 2008.

[86] S. Siegel, *Non-parametric statistics for the behavioral sciences*, McGraw-Hill, 1956.

[87] S. Sinha and M. J. Harrold, "Analysis and Testing of Programs with Exception Handling Constructs," *IEEE Transactions on Software Engineering*, vol. 26 (9), pp. 849-871, 2000.

[88] L. E. Toothaker, *Introductory statistics for the behavioral sciences*, Mcgraw-Hill College, Second Edition, 1986.

[89] R. J. Trent, *Clinical Bioinformatics*, Springer, 2007.

[90] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*, Morgan-Kaufmann, 2006.

[91] E. Weyuker, "The Cost of Data Flow Testing: An Empirical Study," *IEEE Transactions of Software Engineering*, vol. 16 (2), pp. 121-128, 1990.

[92] E. Weyuker, T. Goradia and A. Singh, "Automatically generating test data from a Boolean specification," *IEEE Transactions on Software Engineering*, vol. 20 (5), pp. 353-363, 1994.

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

[94] N. Yevtushenko, A. Lebedev and A. Petrenko, "On the checking experiments with nondeterministic automata," *Automatic control and computer sciences*, vol. 25 (6), 1991.

Appendix A Class Clusters State machines and class diagrams

A.1 Cruise Control Class Diagram

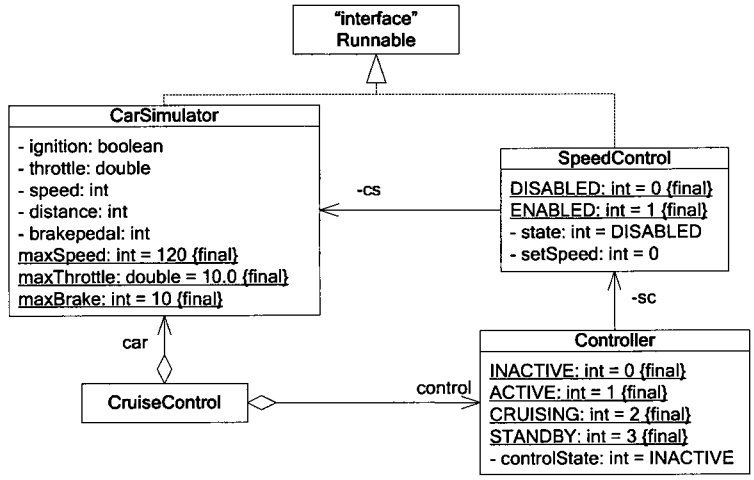


Figure 31: Cruise Control's class diagram

A.2 Cruise Control State machine

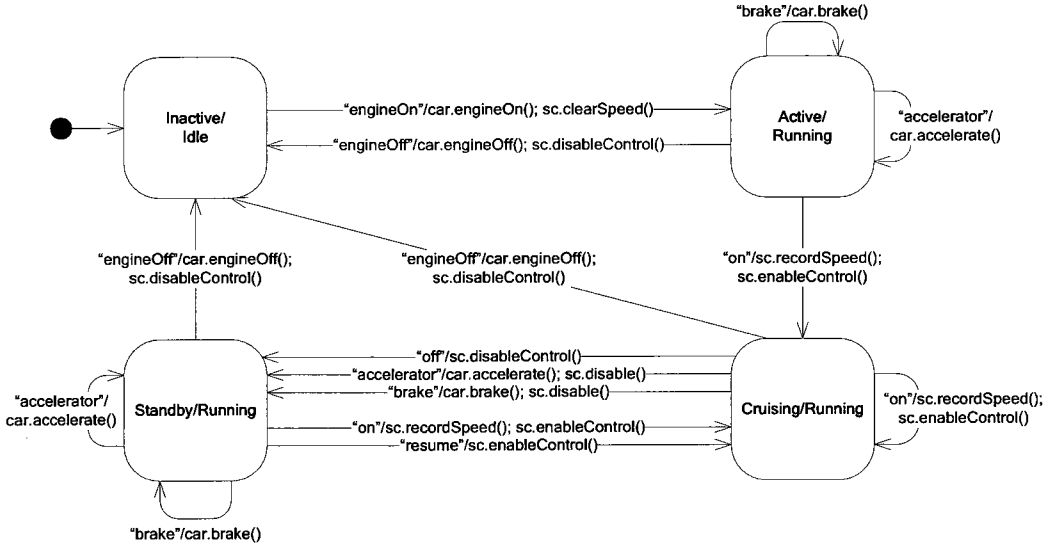


Figure 32: Cruise Control state machine (Carleton 1 experiment)

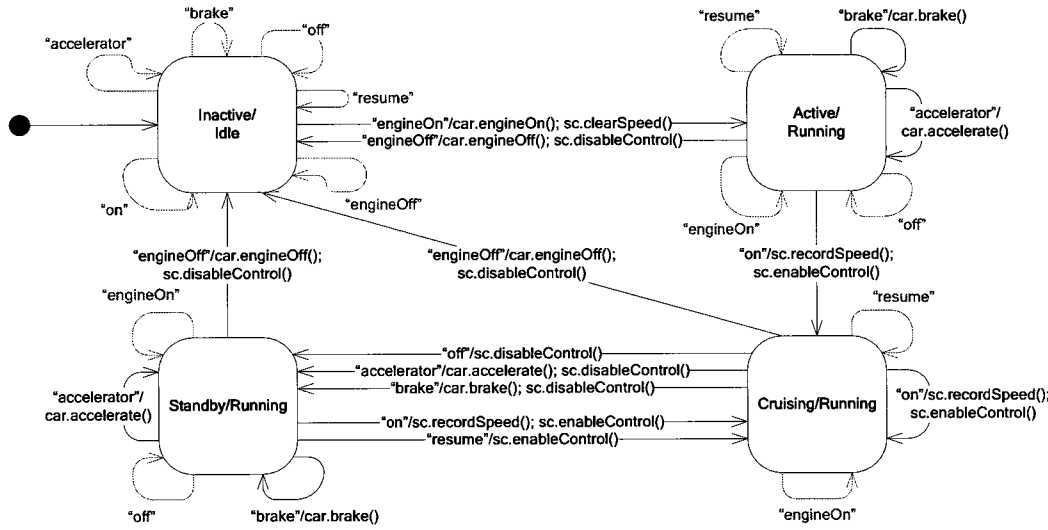
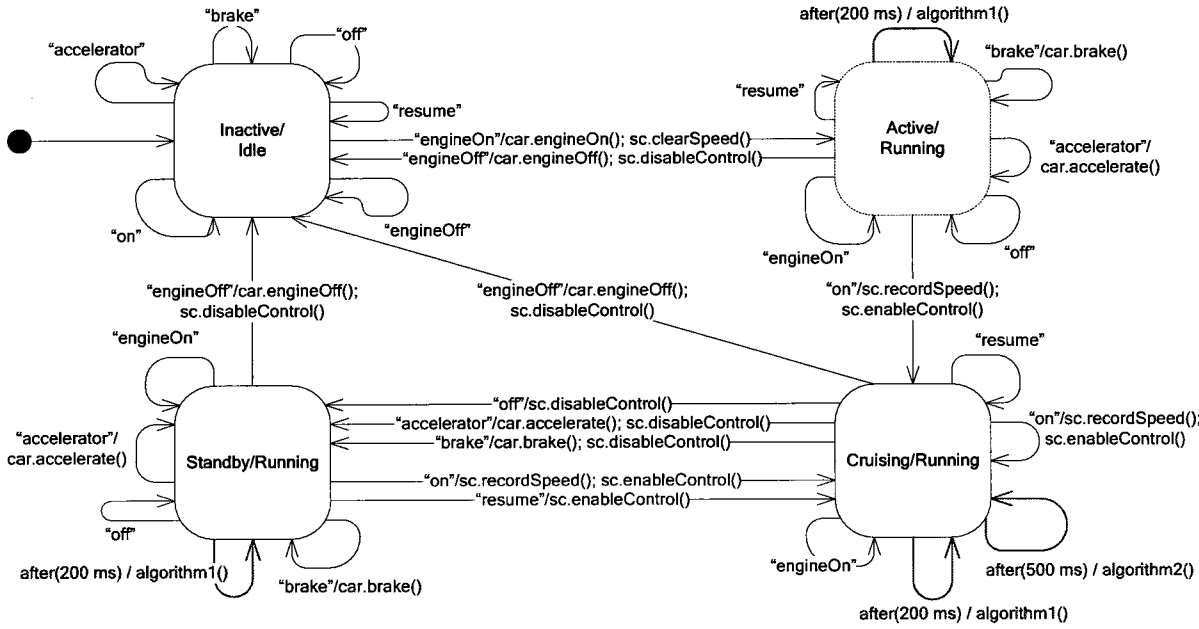


Figure 33: Cruise Control state machine (Replication experiments)



Algorithm 1:

```
fspeed = fspeed+((throttle - fspeed/airResistance - 2*brakepedal))/ticksPerSecond;
if (fspeed>maxSpeed) fspeed=maxSpeed;
if (fspeed<0) fspeed=0;
fdist = fdist + (fspeed/36.0)/ticksPerSecond;
speed = (int)fspeed;
distance=(int)fdist;
if (throttle>0.0) throttle-=0.5/ticksPerSecond; if (throttle<0.0) throttle=0;
```

Algorithm 2:

```
double error = (float)(setSpeed-cs.getSpeed())/6.0;
double steady = (double)setSpeed/12.0;
cs.setThrottle(steady+error);
```

Figure 34: Cruise Control's state machine mdeling real-time behavior

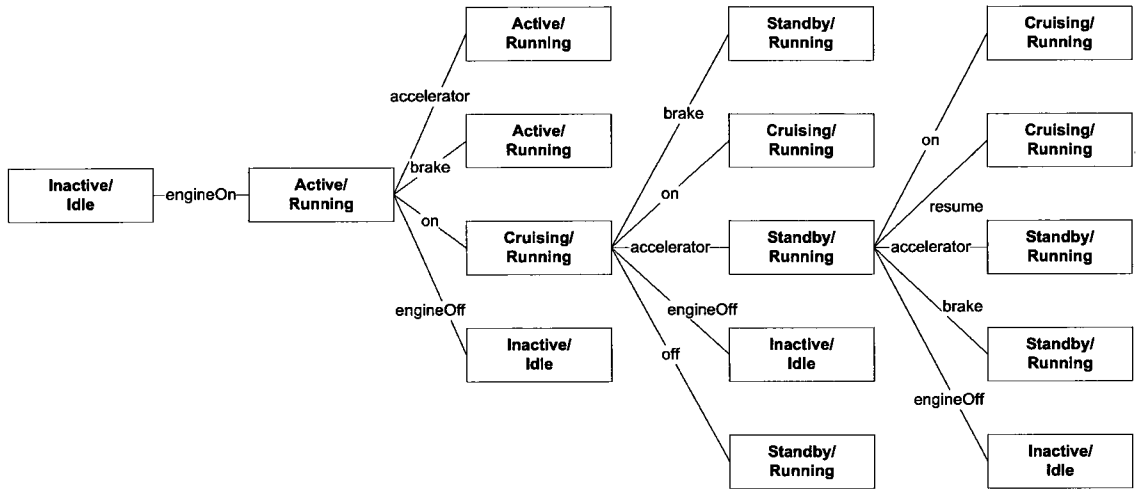


Figure 35: Cruise Control's transition tree (Carleton 1 experiment)

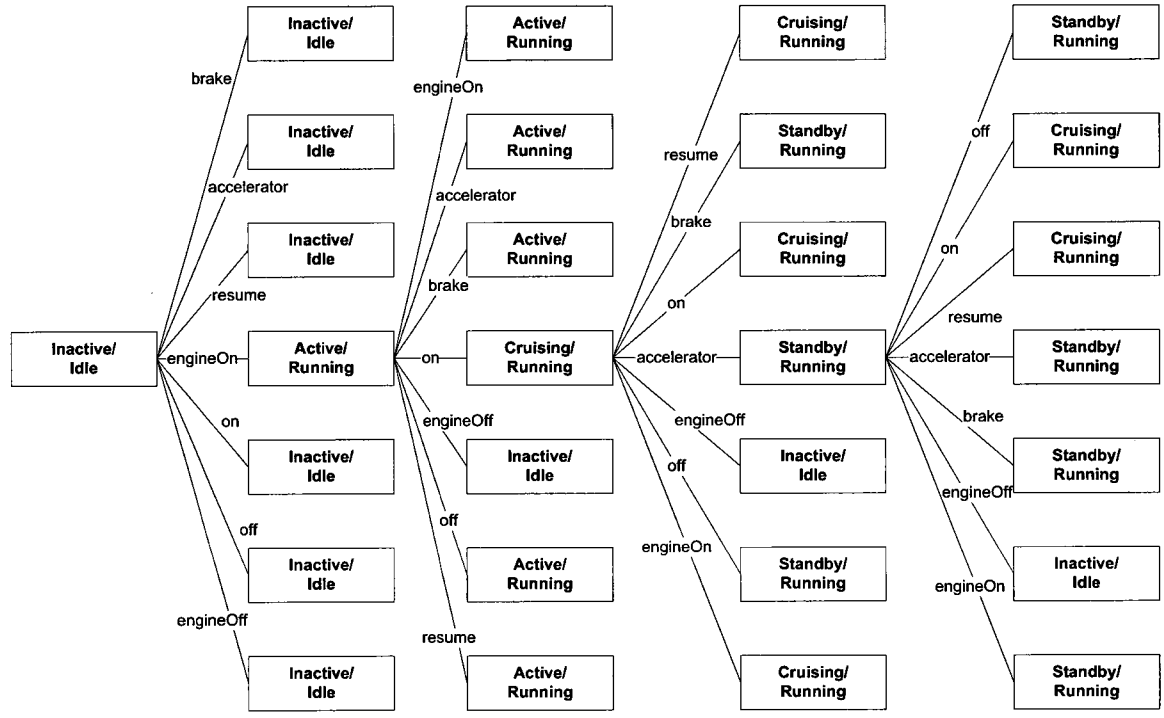


Figure 36: Breadth-first traversal Cruise Control's transition tree

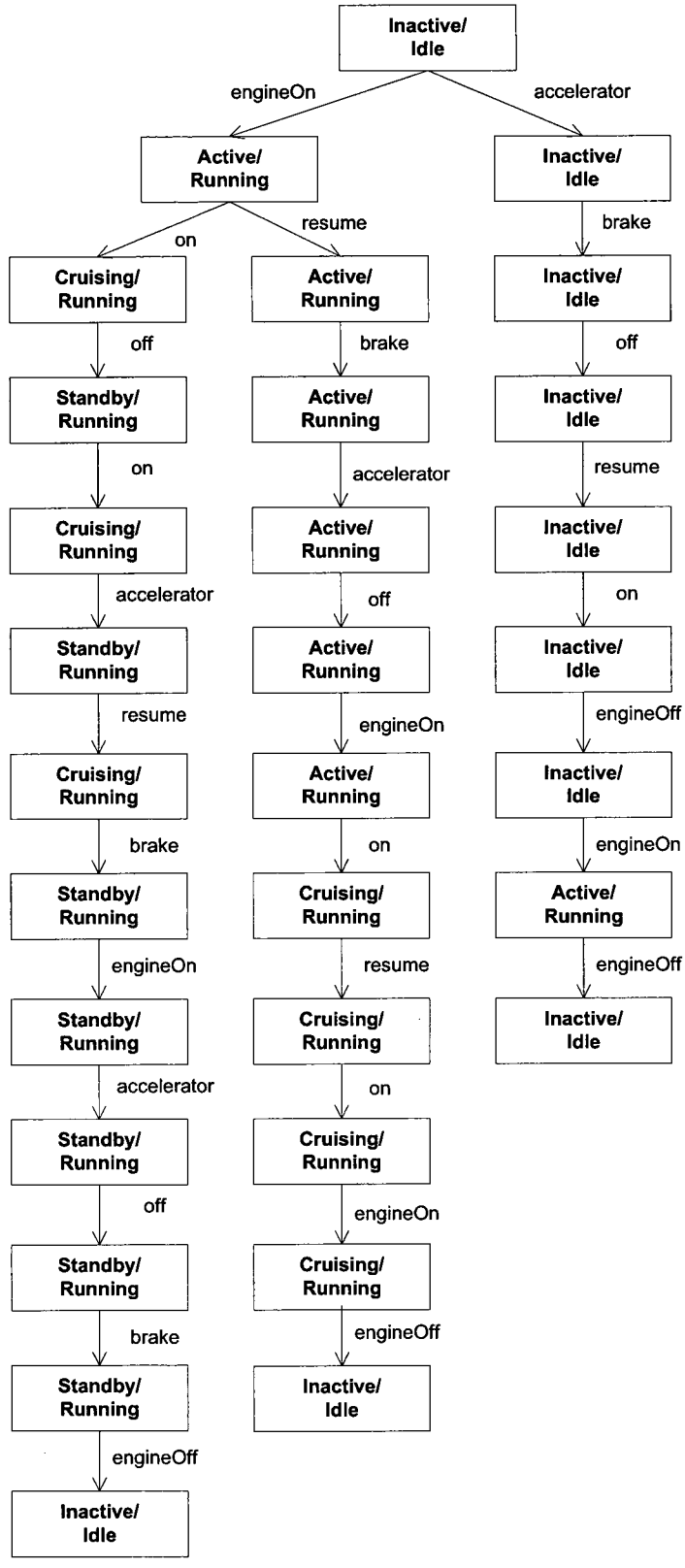


Figure 37: Depth-first search traversal Cruise Control's transition tree

A.3 `OrdSet` Class Diagram

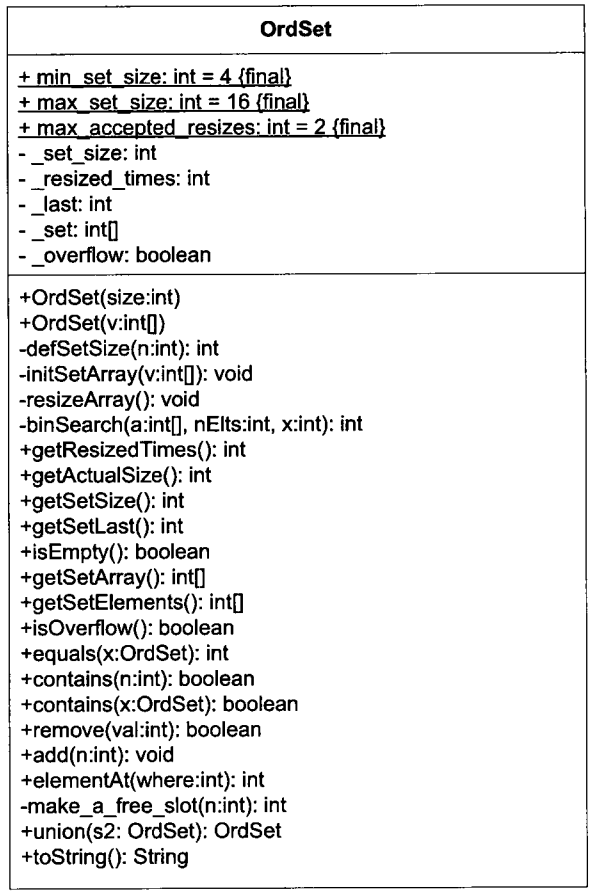


Figure 38: OrdSet's class diagram

A.4 `OrdSet` State machine

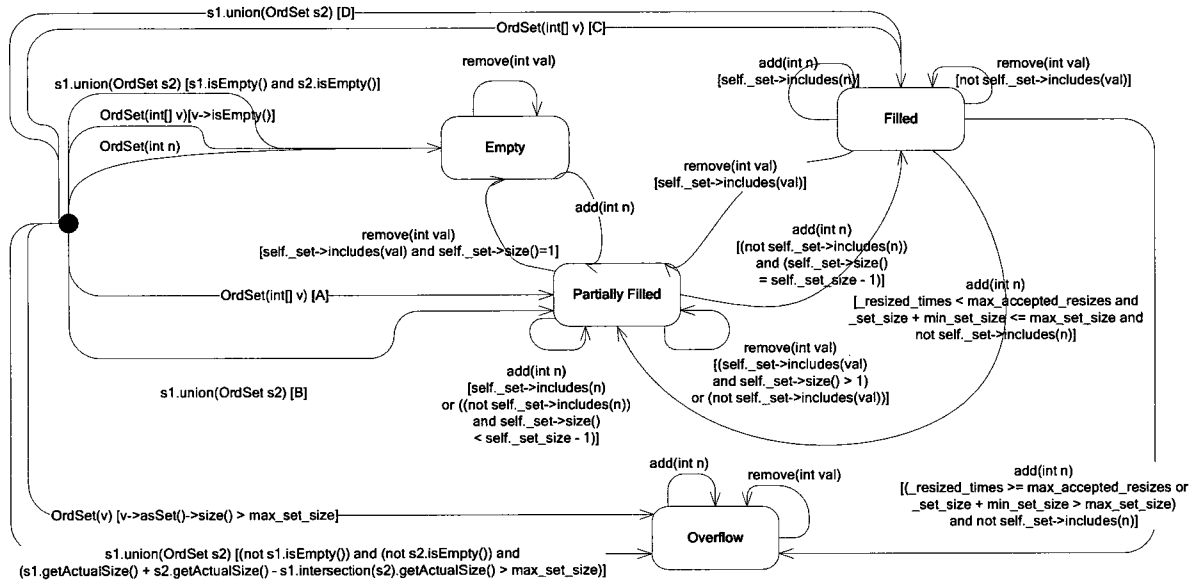


Figure 39: OrdSet's state machine

Conditions

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

B:  (s1.isEmpty()) and (not s2.isEmpty())
    and (s2.getActualSize().mod(min_set_size) <> 0)
    or  (not s1.isEmpty()) and (s2.isEmpty())
        and (s1.getActualSize().mod(min_set_size) <> 0)
    or  (not s1.isEmpty()) and (not s2.isEmpty())
        and (s1.intersection(s2).isEmpty())
        and (s1.getActualSize() + s2.getActualSize() < max_set_size)
        and ((s1.getActualSize() + s2.getActualSize()).mod(min_set_size) <> 0)
    or  (not s1.isEmpty()) and (not s2.isEmpty())
        and (not s1.intersection(s2).isEmpty())
        and (s1.getActualSize() + s2.getActualSize() <= max_set_size)
    or  (not s1.isEmpty()) and (not s2.isEmpty())
        and (not s1.intersection(s2).isEmpty())
        and (s1.getActualSize() + s2.getActualSize() > max_set_size)
        and (s1.getActualSize() + s2.getActualSize() -
             s1.intersection(s2).getActualSize() < max_set_size)
```

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

D:  (s1.isEmpty()) and (not s2.isEmpty())
    and (s2.getActualSize().mod(min_set_size) = 0)
    or  (not s1.isEmpty()) and (s2.isEmpty())
        and (s1.getActualSize().mod(min_set_size) = 0)
    or  (not s1.isEmpty()) and (not s2.isEmpty())
        and (s1.intersection(s2).isEmpty())
        and (s1.getActualSize() + s2.getActualSize() = max_set_size)
    or  (not s1.isEmpty()) and (not s2.isEmpty())
        and (s1.intersection(s2).isEmpty())
        and (s1.getActualSize() + s2.getActualSize() < max_set_size)
        and ((s1.getActualSize() + s2.getActualSize()).mod(min_set_size) = 0)
    or  (not s1.isEmpty()) and (not s2.isEmpty())
        and (not s1.intersection(s2).isEmpty())
        and (s1.getActualSize() + s2.getActualSize() > max_set_size)
        and (s1.getActualSize() + s2.getActualSize()
            - s1.intersection(s2).getActualSize() = max_set_size)
```

Path conditions:

[1]: s1.getSetElements()->isEmpty() and s2.getSetElements()->isEmpty()
[2]: v->isEmpty()
[3]: v->notEmpty() and ((v->size()).mod(min_set_size) = 0 and v->asSet()->size() < v->size())
[4]: v->notEmpty() and (v->size() > max_set_size and v->asSet()->size() < max_set_size)
[5]: v->notEmpty() and ((v->size()).mod(min_set_size)<> 0)
[6]: s1.getSetElements()->notEmpty() and s2.getSetElements()->notEmpty() and (s1.getSetElements()->size() + s2.getSetElements()->size()).mod(min_set_size) = 0
and s1.getSetElements()->intersection(s2.getSetElements())->size() <> 0)
[7]: s1.getSetElements()->notEmpty() and s2.getSetElements()->isEmpty()
and (s1.getSetElements()->size() + s2.getSetElements()->size()).mod(min_set_size) <> 0)
[8]: s1.getSetElements()->notEmpty() and s2.getSetElements()->isEmpty()
and (s1.getSetElements()->union(s2.getSetElements())->size() < max_set_size
[9]: s2.getSetElements()->notEmpty() and s1.getSetElements()->isEmpty()
and (s1.getSetElements()->size() + s2.getSetElements()->size()).mod(min_set_size) <> 0)
[10]: s2.getSetElements()->notEmpty() and s1.getSetElements()->isEmpty()
and (s1.getSetElements()->union(s2.getSetElements())->size() < max_set_size
[11]: (s1.getSetElements()->notEmpty() and s2.getSetElements()->notEmpty())
and (s1.getSetElements()->size() + s2.getSetElements()->size()).mod(min_set_size) = 0
and s1.getSetElements()->intersection(s2.getSetElements())->size() = 0)
[12]: (s1.getSetElements()->notEmpty() and s2.getSetElements()->notEmpty())
and (s1.getSetElements()->union(s2.getSetElements())->size() = max_set_size)
[13]: v->notEmpty() and (v->size() > max_set_size and v->asSet()->size() = max_set_size)
[14]: v->notEmpty() and ((v->size()).mod(min_set_size) = 0 and v->asSet()->size() = v->size())

[15]: s1.getSetElements()->notEmpty() and s2.getSetElements()->notEmpty()
and s1.getSetElements()->union(s2.getSetElements())->size() > max_set_size
[16]: self.getSetElements()->includes(val) and self.getSetElements()->size()=1
[17]: self.getSetElements()->includes(n)
[18]: (not self.getSetElements()->includes(n)) and self.getSetElements()->size() < self.getSetElements()_size – 1
[19]: self.getSetElements()->includes(val) and self.getSetElements()->size() > 1
[20]: not self.getSetElements()->includes(val)
[21]: (not self.getSetElements()->includes(n)) and (self.getSetElements()->size() = self.getSetElements()_size – 1)
[22]: not self.getSetElements()->includes(val)
[23]: _resized_times < max_accepted_resizes and _set_size + min_set_size <= max_set_size
and not self.getSetElements()->includes(n)
[24]: (_set_size + min_set_size > max_set_size) and (not self.getSetElements()->includes(n))
[25]: self.getSetElements()->includes(n)
[26]: self.getSetElements()->includes(val)
[27]: (_resized_times >= max_accepted_resizes) and (not self.getSetElements()->includes(n))
[28]: v->asSet()->size() > max_set_size

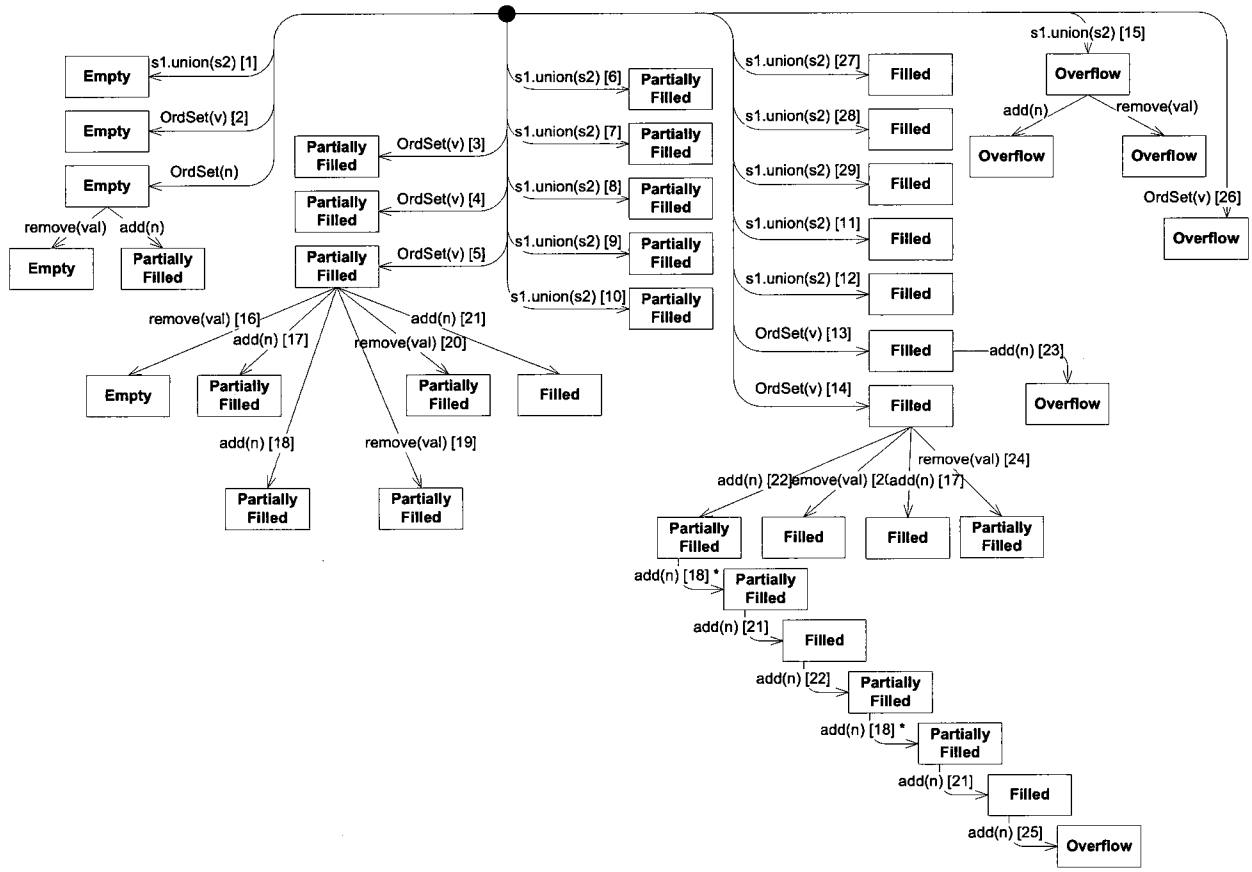


Figure 40: Breadth-first search traversal OrdSet's transition tree

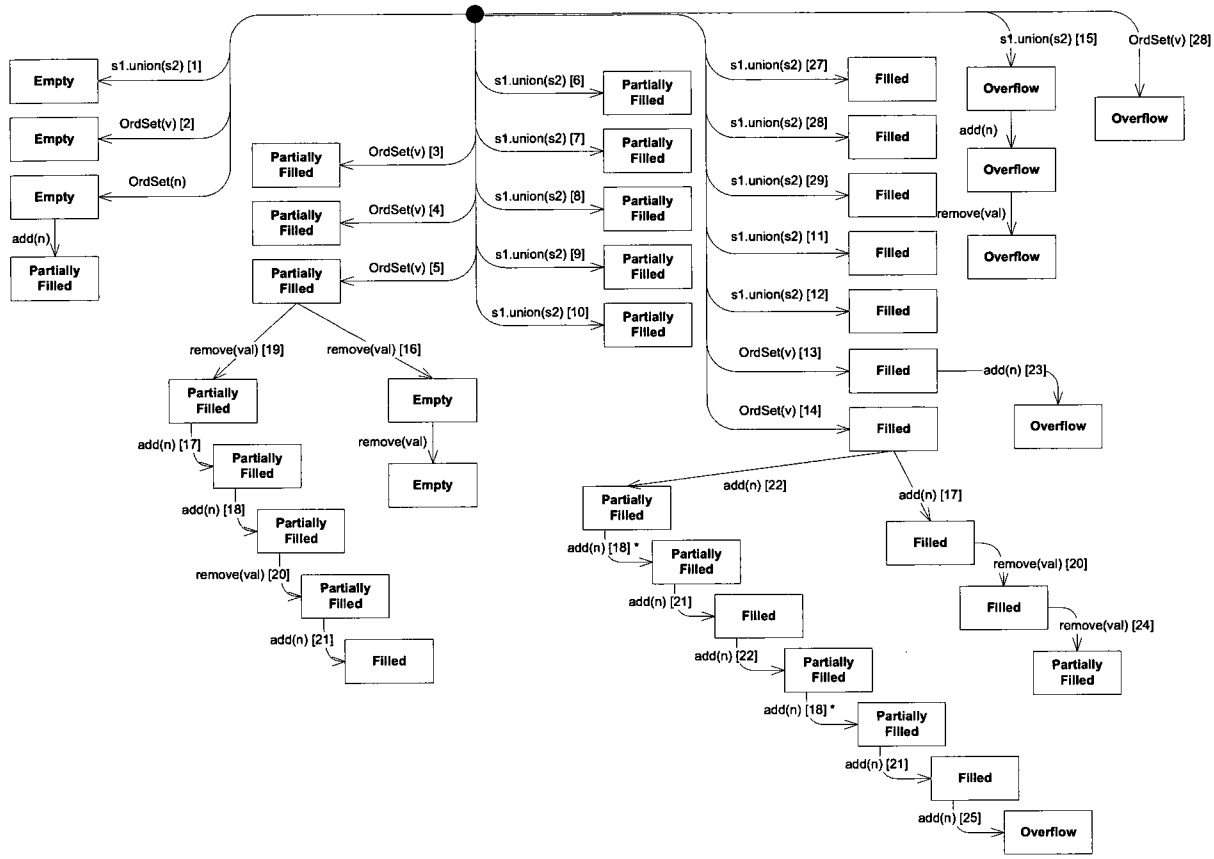


Figure 41: Depth-first search traversal OrdSet's transition tree

A.5 Elevator Class diagram

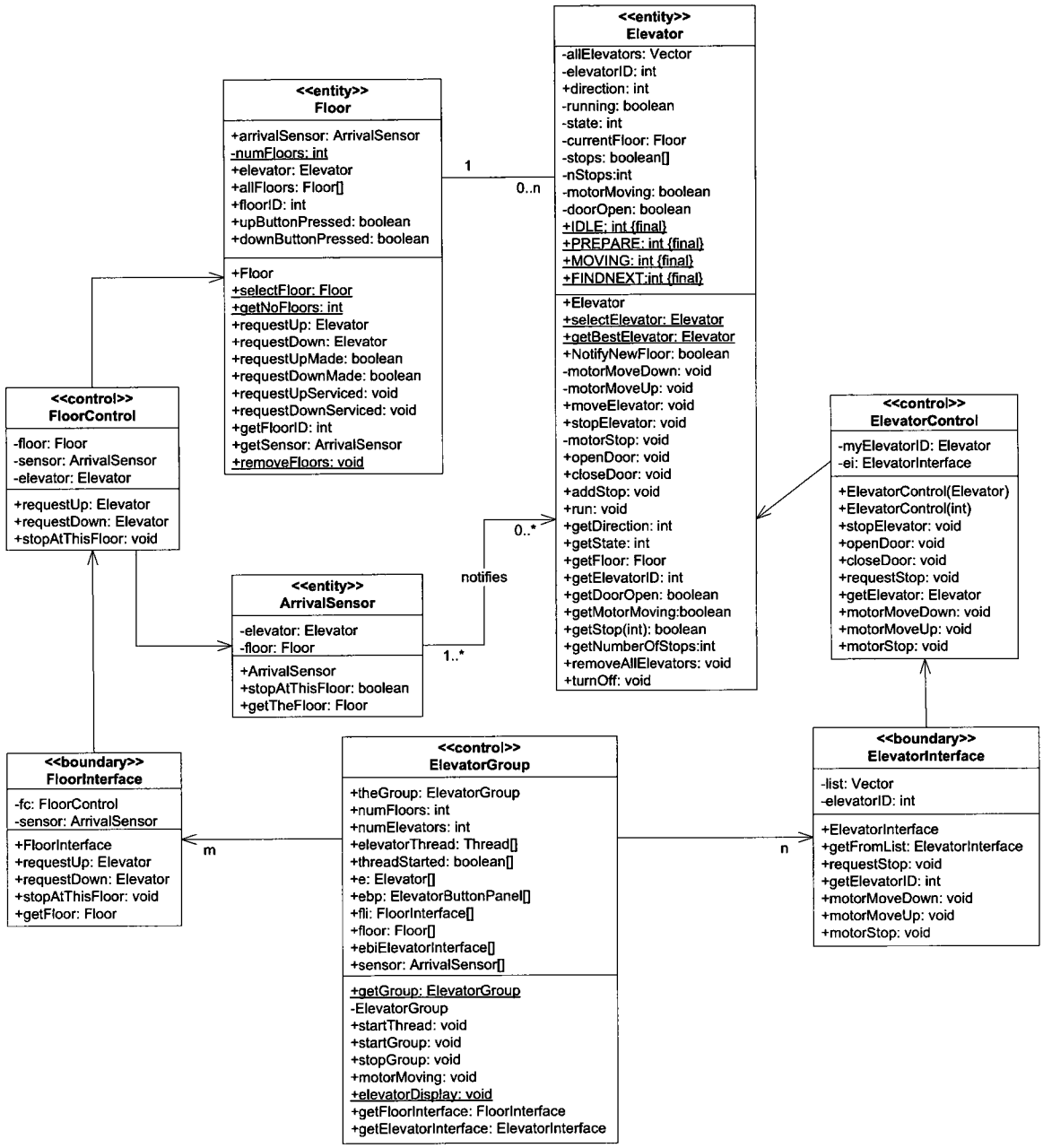


Figure 42: Elevator's class diagram

A.6 Elevator state machine

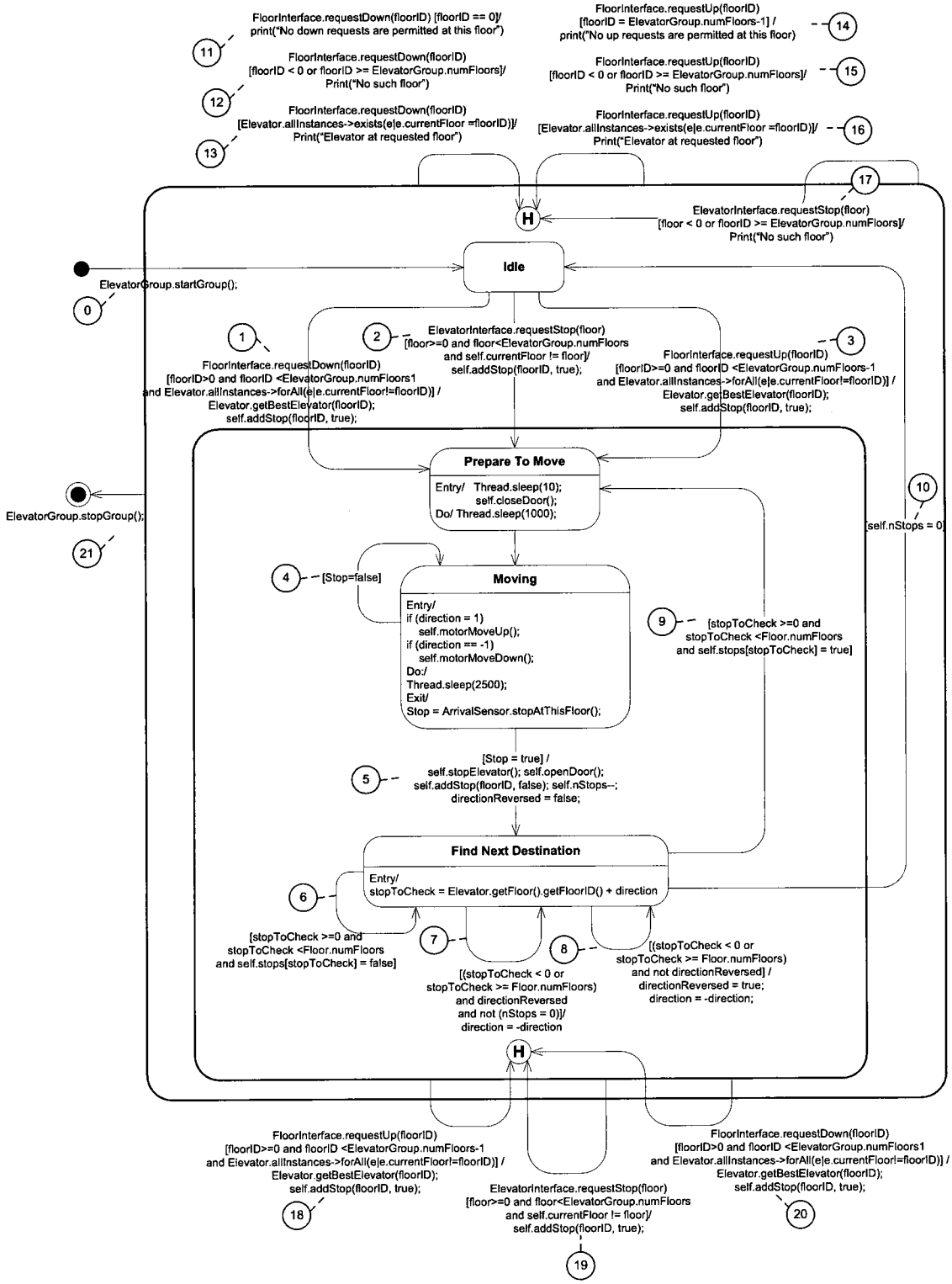


Figure 43: Elevator class's state machine

1 – Starting the elevator system by starting its threads.

2 – Requesting an elevator from floor to go down (No elevator is already on floor).

3 – Requesting a stop from an elevator (Elevator is on a different floor than the one requested).

4 – Requesting an elevator from floor to go up (No elevator is already on floor).

5 – Door is completely closed.

6 – No requested stop at reached floor.

7 – A requested stop exists for the reached floor.

8 – A requested stop not serviced yet exists (next destination to service).

9 – No more requested stops for the elevator.

10 – Down request at first floor.

11 – Down request from a non recognized floor (wrong ID).

12 – Down request from floor where an elevator has this floor as current floor (i.e stopped at or moving from).

13 – Request a stop from elevator for a non recognized floor number.

14 – Up request at last floor.

15 – Up request from a non recognized floor (wrong ID).

16 – Up request from floor where an elevator has this floor as current floor (i.e stopped at or moving from).

17 – Requesting an elevator from floor to go up (No elevator is already on floor).

18 – Requesting an elevator from floor to go down (No elevator is already on floor).

19 – Requesting a stop from an elevator (Elevator is on a different floor than the one requested).

20 – Shutdown the elevator system.

A.7 Elevator's transition tree

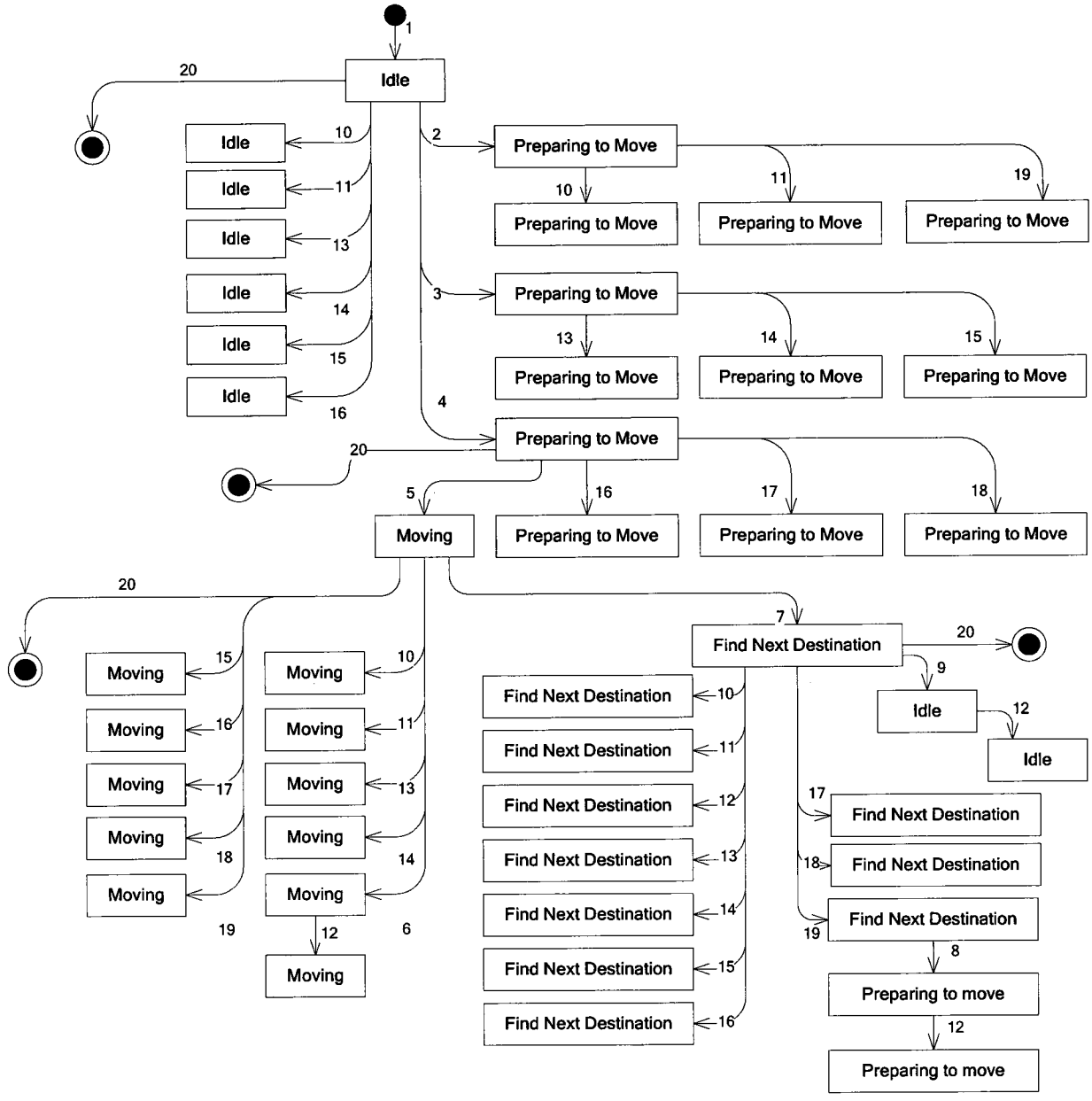


Figure 44: Breadth-first search traversal Elevator's transition tree

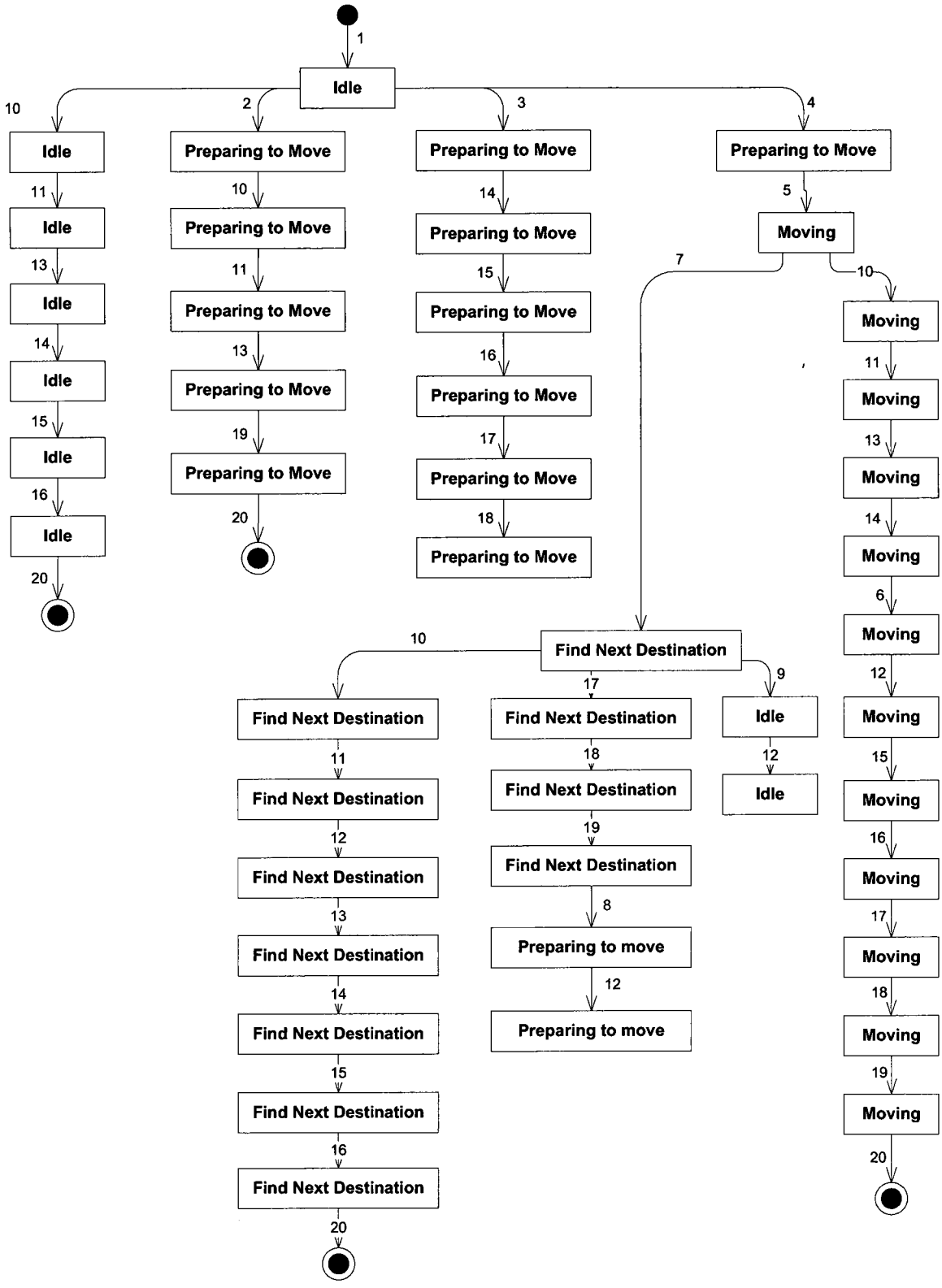


Figure 45: Depth-first search traversal Elevator's transition tree

Appendix B State Invariants

B.1 Cruise Control's state invariants

B.1.1 Inactive/Idle State

```
control.controlState = INACTIVE
and  control.sc.state = DISABLED  and  control.sc.cs.speed = 0
and  not control.sc.cs.ignition  and  control.sc.cs.throttle = 0
and  control.sc.cs.distance = 0  and  control.sc.cs.brakepedal = 0
```

B.1.2 Active/Running State

```
control.controlState = ACTIVE  and  control.sc.state = DISABLED
and  control.sc.setSpeed = 0  and  control.sc.cs.ignition
and  control.sc.cs.distance >= 0
```

B.1.3 Cruising/Running State

```
control.controlState = CRUISING  and  control.sc.state = ENABLED
and  control.sc.setSpeed >= 0  and  control.sc.cs.ignition
```

B.1.4 Standby/Running State

```
control.controlState = STANDBY  and  control.sc.state = DISABLED
and  control.sc.setSpeed >= 0  and  control.sc.cs.ignition
and  control.sc.cs.distance = 0
```

B.2 OrdSet's state invariants

B.2.1 Empty state invariant:

```
_resized_times <= max_accepted_resizes  and  _last = -1
and  self.getActualSize() = 0  and  self.getSetElements()->size() = 0
and  _overflow = false
```

B.2.2 Partially Filled state invariant:

```
_resized_times <= max_accepted_resizes
and  _last >= 0 and _last < _set_size - 1
and  self.getActualSize() < _set_size and _set_size > 0
and  _overflow = false
```

B.2.3 Filled state invariant:

```
_resized_times <= max_accepted_resizes  and  _last = _set_size - 1
and  _set->size() = _set_size  and  _overflow = false
```

B.2.4 Overflow state invariant:

```
(_resized_times = max_accepted_resizes or _set_size ==  max_set_size)
and  _last = _set_size - 1  and  _set->size() = _set_size
and  _overflow = true
```

B.3 Elevator's state invariants

B.3.1 Idle State

```
Sequence{0 .. ElevatorGroup.numFloors-1} -> forAll(i|not
Elevator.stops[i])
and  Elevator.state = IDLE  and  Elevator.doorOpen
and  not Elevator.motorMoving  and  direction = 0
and  nStops = 0
```

B.3.2 Preparing to Move State

```
Sequence{0 .. ElevatorGroup.numFloors-1} -> exists(i|
 Elevator.stops[i])
and  Elevator.state = PREPARE  and  not Elevator.doorOpen
and  not Elevator.motorMoving  and  (direction = 1 or direction = -1)
and  nStops > 0
```

B.3.3 Moving State

```
Sequence{0 .. ElevatorGroup.numFloors-1} -> exists(i|
 Elevator.stops[i])
and  Elevator.state = MOVING  and  not Elevator.doorOpen
and  Elevator.motorMoving  and  (direction = 1 or direction = -1)
and  nStops > 0
```

B.3.4 Find Next Destination State

```
Elevator.state = FINDNEXT  and Elevator.doorOpen
and not Elevator.motorMoving  and (direction = 1 or direction = -1)
```

Appendix C Mutation Operators

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

Table 51: Mutation operators' descriptions

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

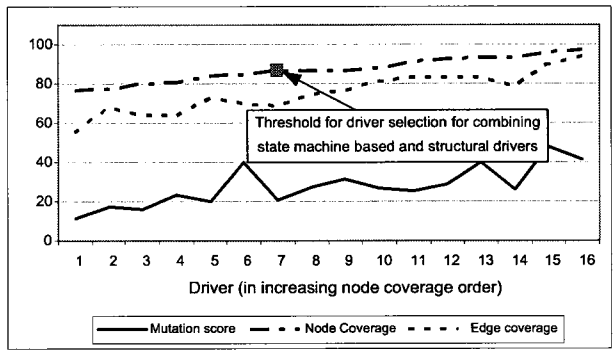


Figure 46: Node and Edge coverage distributions (Cruise Control – Carleton 1)

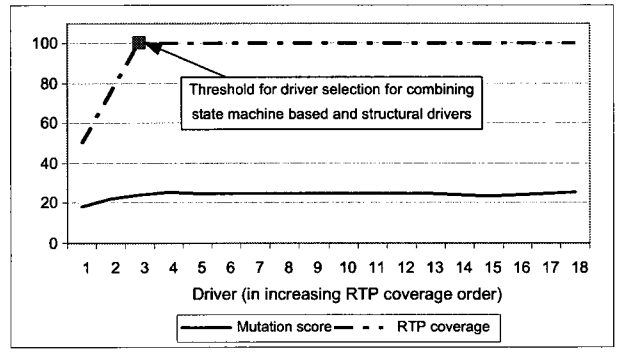


Figure 47: RTP coverage distribution (Cruise Control – Carleton 1)

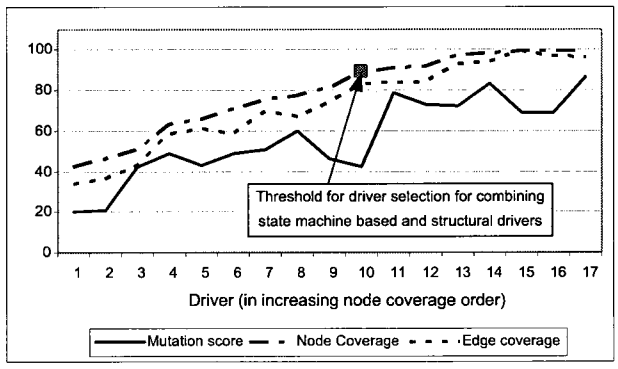


Figure 48: Node and Edge coverage distributions (OrdSet – Carleton 1)

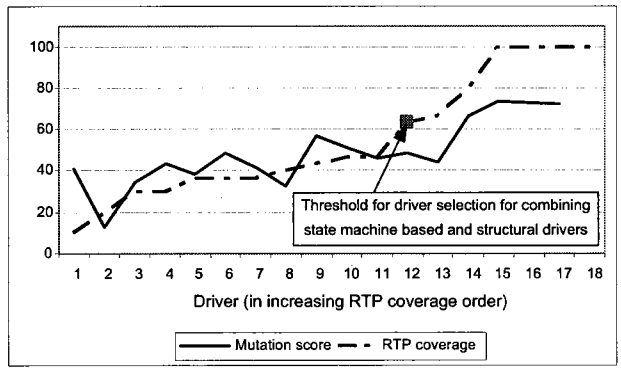


Figure 49: RTP coverage distribution (OrdSet – Carleton 1)

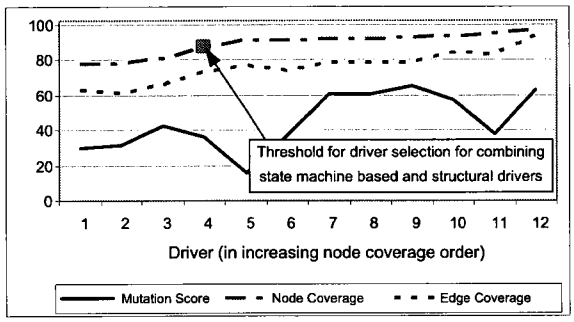


Figure 50: Node and Edge coverage distributions (Cruise Control – Sannio 1)

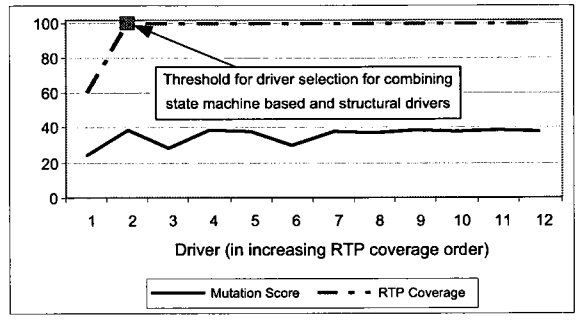


Figure 51: RTP coverage distribution (Cruise Control – Sannio 1)

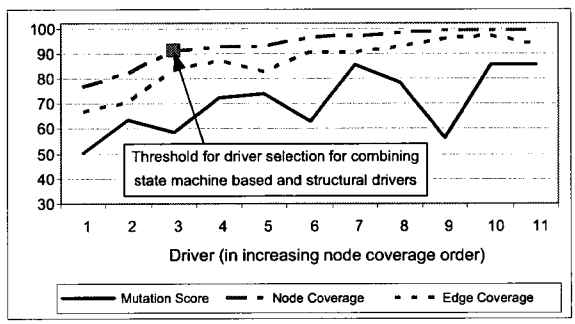


Figure 52: Node and Edge coverage distributions (OrdSet – Sannio 1)

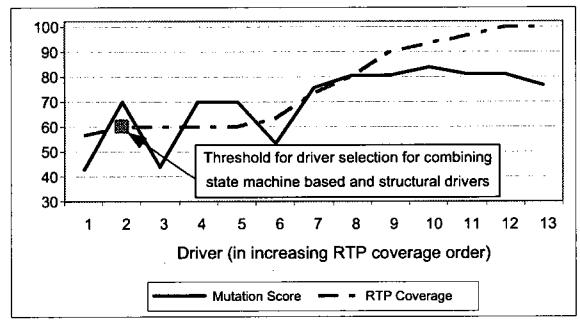


Figure 53: RTP coverage distribution (OrdSet – Sannio 1)

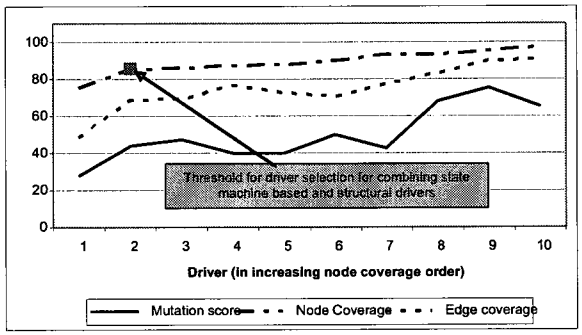


Figure 54: Node and Edge coverage distributions (Cruise Control – Carleton 2)

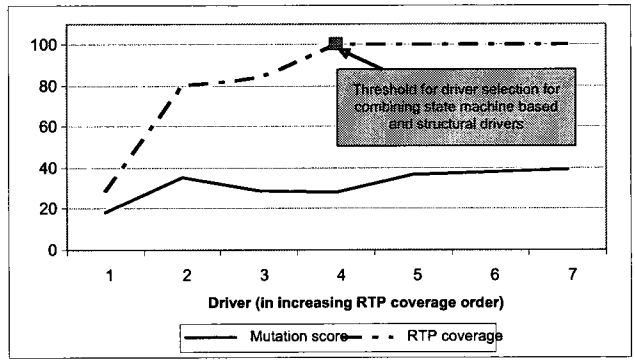


Figure 55: RTP coverage distribution (Cruise Control – Carleton 2)

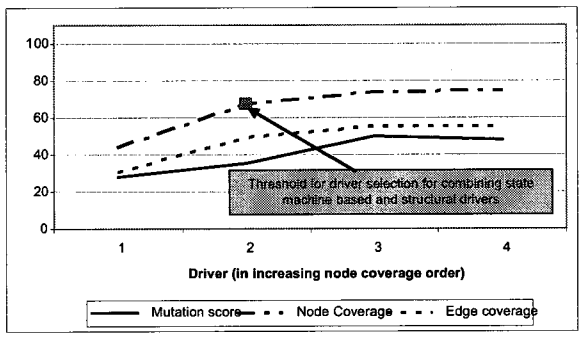


Figure 56: Node and Edge coverage distributions (Elevator – Carleton 2)

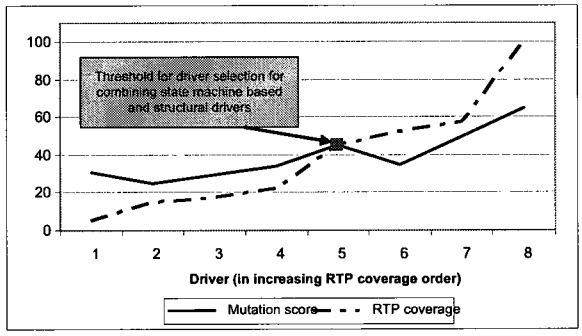


Figure 57: RTP coverage distribution (Elevator – Carleton 2)

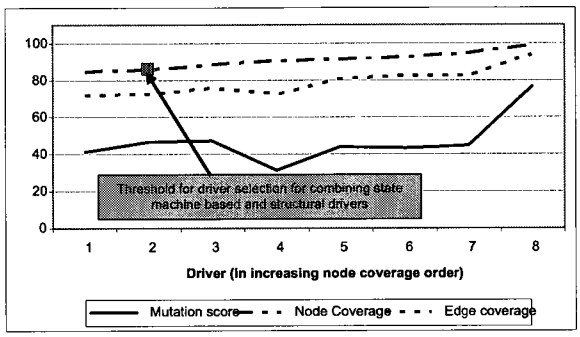


Figure 58: Node and Edge coverage distributions (Cruise Control – Sannio 2)

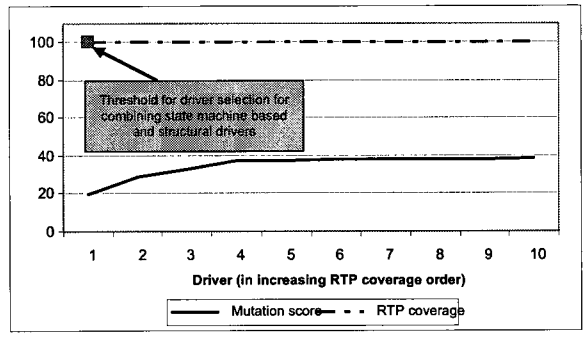


Figure 59: RTP coverage distribution (Cruise Control – Sannio 2)

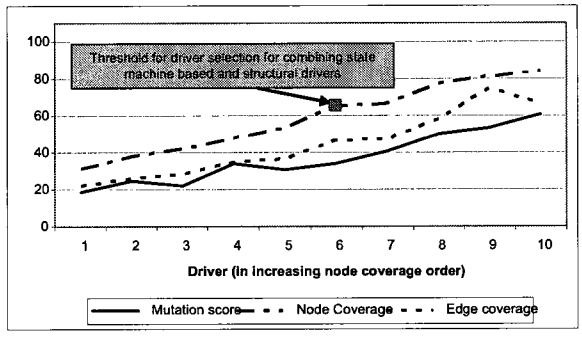


Figure 60: RTP coverage distribution (Elevator – Sannio 2)

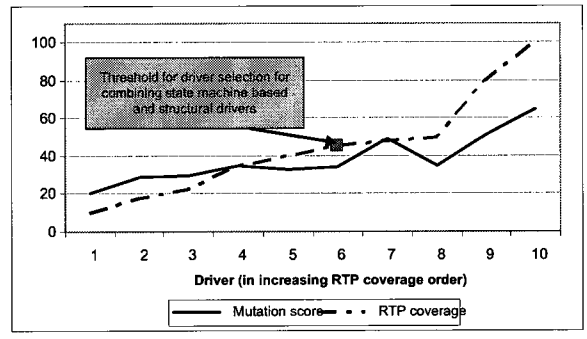


Figure 61: RTP coverage distribution (Elevator – Sannio 2)

Appendix E Descriptive statistics tables

E.1 Mutation scores descriptive statistics

Cluster	Experiment	Treatment	Min	Max	Mean	Std Dev	Range
OrdSet	Carleton 1	State machine	12.66	79.17	50.27	17.2	66.51
		Structural	20.35	86.7	56.15	19.98	66.35
	Sannio 1	State machine	42.79	83.81	*71.76*	12.41	41.03
		Structural	50.32	85.74	70.31	12.69	35.42
Cruise Control	Carleton 1	State machine	18.39	27.46	24.47	1.65	9.07
		Structural	11.4	48.19	27.69	10.24	36.79
	Sannio 1	State machine	24.87	39.01	35.65	4.83	14.14
		Structural	15.45	65.18	44.66	16.05	49.74
	Carleton 2	State machine	18.06	37.7	30.8	7.43	19.63
		Structural	28.01	75.13	50.13	14.81	47.12
	Sannio 2	State machine	19.9	38.48	34.55	6.07	18.59
		Structural	31.15	76.96	46.89	18.17	45.81
Elevator	Carleton 2	State machine	24.83	49.66	35.45	8.84	24.83
		Structural	28.06	50.17	40.54	10.53	22.11
	Sannio 2	State machine	20.58	51.02	35.06	9.62	30.44
		Structural	18.62	60.71	36.97	14.1	42.09

Table 52: Mutation scores descriptive statistics

E.2 Node, edge and RTP coverage descriptive statistics

Cluster	Experiment	Treatment	Coverage	Median	Mean	Variance	Range	Min	Max
OrdSet	Carleton 1	State machine	RTP	45	34.81	798.51	90	10	100
			Node	71.17	76.93	106.45	30.36	63.96	94.59
			Edge	61.9	67.5	163	42.07	48.41	90.48
		Structural	Node	81.08	81.08	333.88	58.65	40.54	99.1
			Edge	73.02	73.53	405.1	65.08	30.95	96.03
	Sannio 1	State machine	RTP	76.67	77.78	305.72	43.33	56.67	100
			Node	90.09	89.11	26.18	17.12	77.48	94.59
			Edge	81.75	80.29	56.33	25.4	65.08	90.48
		Structural	Node	96.4	93.12	56.37	22.52	76.58	99.1
			Edge	90.48	86.44	99.2	30.16	66.67	96.83
Cruise Control	Carleton 1	State machine	RTP	100	96.59	184.9	50	50	100
			Node	85.85	84.79	14.62	16.01	78.3	94.34
			Edge	69.9	68.93	29.09	24.36	58.25	79.61
		Structural	Node	86.79	87.92	36.26	19.61	76.42	97.17
			Edge	75.73	75.61	83.95	30.09	55.34	94.17
	Sannio 1	State machine	RTP	100	96.67	133.33	40	60	100
			Node	85.37	85.76	4.35	7.34	84.47	91.8
			Edge	75.42	76.42	6.38	9.34	74.05	83.39
		Structural	Node	89.49	87.98	38.6	20	77.3	97.3
			Edge	75.75	74.57	79.11	29.7	63.87	93.58
	Carleton 2	State machine	RTP	100	84.57	779.2	72	28	100
			Node	84.47	81.19	59.75	20.13	65.24	85.37
			Edge	74.96	70.03	119.66	28.8	47.99	76.78
		Structural	Node	88.97	89.18	41	22.1	75.2	97.3
			Edge	74.75	74.78	149.74	42.2	48.46	90.66
	Sannio 2	State machine	RTP	100	100	0	0	100	100
			Node	85.85	86.04	3.91	6.6	84.91	91.51
			Edge	76.7	77.19	4.24	6.8	75.73	82.52
		Structural	Node	91.04	90.92	21.09	14.15	84.91	99.06
			Edge	78.15	79.12	56.55	22.33	71.84	94.17
Elevator	Carleton 2	State machine	RTP	22.5	30.71	424.4	52.5	5	57.5
			Node	65.15	62.81	129.4	31.95	41.08	73.03
			Edge	47.66	45.45	30.17	26.23	29.84	56.07
		Structural	Node	70.74	65.04	207	30.71	43.98	74.69
			Edge	52.11	47.66	134.65	24.77	30.84	55.61
	Sannio 2	State machine	RTP	40	38.61	437.67	70	10	80
			Node	55.19	56.84	150.56	41.49	33.2	74.69
			Edge	42.06	43.41	106.2	35.05	23.36	58.41
		Structural	Node	59.13	58.75	359.81	53.11	31.12	84.23
			Edge	41.85	44.16	316.29	52.8	21.96	74.77

Table 53: Structural and state machine coverage descriptive statistics

E.3 Number of method calls descriptive statistics

	Cluster	Treatment	Min	Max	Mean	StdDev	Range
Carleton 1	**OrdSet**	State machine	73	2732	875.55	781.48	2659
		Structural	84	1350	600.64	409.26	1266
	Cruise Control	State machine	230	683	502.17	114.60	453
		Structural	39	1155	382.43	310.08	1116
Sannio 1	**OrdSet**	State machine	525	2417	1732.33	658.38	1892
		Structural	214	1783	787.81	425.32	1569
	Cruise Control	State machine	550	1543	1298.5	325.33	993
		Structural	96	557	246.83	123.55	461
Carleton 2	**Elevator**	State machine	286	7686	1719.75	2472.515	1490
		Structural	307	591	455.25	121.72	284
	Cruise Control	State machine	330	1539	1130.43	418.62	1209
		Structural	153	2133	684	643.87	1980
Sannio 2	**Elevator**	State machine	312	7686	1740.4	2176.05	2145
		Structural	85	2696	728.7	801.07	2611
	Cruise Control	State machine	770	1539	1143.18	235.68	720
		Structural	196	639	369.5	141.08	443

Table 54: Cost descriptive statistics

Appendix F Complementariness analysis data

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

Table 55: Mutation scores and difference fault sets scores (Carleton 1)

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

Table 56: Intersection score and ratios (Carleton 1)

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

Table 57: Union and not detected faults scores and ratios (Carleton 1)

	Cruise Control				OrdSet			
	|Fs|	| Fc|	|Fs-Fc| /|F|%	|Fc-Fs| /|F|%	|Fs|	| Fc|	|Fs-Fc| /|F|%	|Fc-Fs| /|F|%
Median	117	229	6.22	29.02	477	462	10.58	9.29
Mean	122	199	6.85	26.69	466	457	12.44	11.11
Min	67	59	0.78	0.00	332	352	1.12	1.76
Max	156	280	25.39	56.48	523	535	31.57	33.81
95%	156	280	16.32	45.60	523	535	27.13	26.23
90%	152	280	12.49	44.09	506	535	26.34	18.43
75%	145	230	8.55	35.88	505	533	22.6	14.74
25%	109	170	3.11	17.49	435	393	2.4	5.29
10%	105	59	2.07	4.25	435	352	2.05	4.62
5%	67	59	1.55	0.26	332	352	1.92	4.12

Table 58: Mutation scores and difference fault sets scores (Sannio 1)

	Cruise Control			OrdSet		
	|Fs∩Fc| /|F|%	|Fs∩Fc| / |Fs|	|Fs∩Fc| / |Fc|	|Fs∩Fc| /|F|%	|Fs∩Fc| / |Fs|	|Fs∩Fc| / |Fc|
Median	25.65	0.80	0.48	61.38	0.86	0.86
Mean	24.66	0.79	0.53	62.17	0.83	0.85
Min	9.07	0.37	0.22	38.14	0.62	0.59
Max	37.56	0.97	0.98	81.25	0.98	0.98
95%	33.97	0.94	0.97	78.69	0.98	0.95
90%	31.66	0.92	0.86	78.53	0.97	0.94
75%	28.89	0.90	0.60	71.63	0.97	0.92
25%	21.63	0.75	0.41	52.97	0.71	0.79
10%	14.77	0.52	0.36	46.96	0.66	0.77
5%	13.47	0.48	0.28	45.51	0.65	0.65

Table 59: Intersection score and ratios (Sannio 1)

	Cruise Control				OrdSet			
	|FsUFc| /|F|%	|F – (FsUFc)| /|F|%	|FsUFc| / |Fs|	|FsUFc| / |Fc|	|FsUFc| /|F|%	|F – (FsUFc)| /|F|%	|FsUFc| / |Fs|	|FsUFc| / |Fc|
Median	62.18	37.82	1.72	1.17	86.7	13.3	1.14	1.12
Mean	58.20	41.80	1.58	1.31	85.71	14.29	1.19	1.16
Min	23.32	24.09	1	1.05	71.15	10.74	1.01	1.02
Max	75.91	76.68	2.43	2.53	89.26	28.85	1.56	1.64
95%	74.90	68.75	2.20	2.47	88.33	20.83	1.48	1.495
90%	74.35	60.41	1.96	1.93	88.14	17.25	1.46	1.26
75%	67.62	49.35	1.8	1.29	87.82	15.3	1.36	1.21
25%	50.65	32.38	1.27	1.1	84.7	12.18	1.03	1.07
10%	39.59	25.65	1.03	1.09	82.75	11.86	1.02	1.06
5%	31.25	25.10	1	1.07	79.17	11.67	1.02	1.05

Table 60: Union and not detected faults scores and ratios (Sannio 1)

	Cruise Control				Elevator			
	|Fs|	| Fc|	|Fs-Fc| /|F|%	|Fc-Fs| /|F|%	|Fs|	| Fc|	|Fs-Fc| /|F|%	|Fc-Fs| /|F|%
Median	143	182	7.85	21.99	557	568	6.72	5.78
Mean	136	201	7.48	24.51	572	526	9.04	6.88
Min	108	153	0.26	10.21	407	419	3.49	1.45
Max	151	287	15.18	46.86	766	590	23.38	16.58
95%	151	287	14.07	39.80	766	590	18.66	15.83
90%	151	274	11.78	38.88	766	590	14.79	14.67
75%	146	249	10.02	32.72	630	590	12.35	8.61
25%	133	163	4.71	15.32	499	419	4.13	3.53
10%	108	153	2.23	12.31	407	419	3.85	1.85
5%	108	153	1.44	10.80	407	419	3.68	1.59

Table 61: Mutation scores and difference fault sets scores (Carleton 2)

	Cruise Control			Elevator		
	|Fs∩Fc| /|F|%	|Fs∩Fc| / |Fs|	|Fs∩Fc| / |Fc|	|Fs∩Fc| /|F|%	|Fs∩Fc| / |Fs|	|Fs∩Fc| / |Fc|
Median	28.27	0.78	0.51	37.76	0.81	0.88
Mean	28.00	0.79	0.55	37.82	0.80	0.85
Min	16.49	0.58	0.35	25.94	0.52	0.67
Max	36.91	0.99	0.75	45.83	0.97	0.96
95%	35.93	0.96	0.74	45.18	0.96	0.95
90%	33.77	0.93	0.70	44.55	0.96	0.95
75%	30.56	0.88	0.65	43.33	0.92	0.91
25%	25.40	0.72	0.48	33.46	0.70	0.81
10%	21.86	0.66	0.42	32.54	0.69	0.69
5%	20.42	0.61	0.38	29.54	0.61	0.68

Table 62: Intersection score and ratios (Carleton 2)

	Cruise Control				Elevator			
	|FsUFc| /|F|%	|F – (FsUFc)| /|F|%	|FsUFc| / |Fs|	|FsUFc| / |Fc|	|FsUFc| /|F|%	|F – (FsUFc)| /|F|%	|FsUFc| / |Fs|	|FsUFc| / |Fc|
Median	58.77	41.23	1.66	1.18	53.75	46.26	1.10	1.17
Mean	59.98	40.02	1.71	1.16	53.74	46.26	1.14	1.22
Min	45.29	23.04	1.25	1.00	43.54	35.12	0.91	1.07
Max	76.96	54.71	2.66	1.32	64.88	56.46	1.56	1.66
95%	76.05	52.09	2.41	1.30	63.90	55.39	1.53	1.47
90%	74.22	51.44	2.08	1.28	62.69	54.34	1.48	1.32
75%	69.76	49.08	1.94	1.25	56.53	49.36	1.22	1.30
25%	50.92	30.24	1.40	1.07	50.64	43.47	0.99	1.08
10%	48.56	25.79	1.32	1.03	45.66	37.31	0.95	1.08
5%	47.91	23.95	1.29	1.02	44.61	36.10	0.93	1.08

Table 63: Union and not detected faults scores and ratios (Carleton 2)

	Cruise Control				Elevator			
	|Fs|	| Fc|	|Fs-Fc| /|F|%	|Fc-Fs| /|F|%	|Fs|	| Fc|	|Fs-Fc| /|F|%	|Fc-Fs| /|F|%
Median	143	170	8.25	16.49	578	586	6.55	5.36
Mean	132	182	8.26	21.31	551	563	8.31	7.25
Min	76	119	0.79	13.09	399	402	1.11	1.11
Max	147	294	20.42	57.07	766	714	20.83	24.57
95%	147	294	19.90	41.10	766	714	20.03	18.91
90%	146	294	17.23	40.31	766	714	16.80	15.65
75%	145	181	8.90	22.97	600	631	11.48	10.20
25%	124	165	6.09	14.40	410	481	2.98	2.64
10%	106	119	2.09	13.61	399	402	2.33	1.60
5%	76	119	2.09	13.21	399	402	1.82	1.33

Table 64: Mutation scores and difference fault sets scores (Sannio 2)

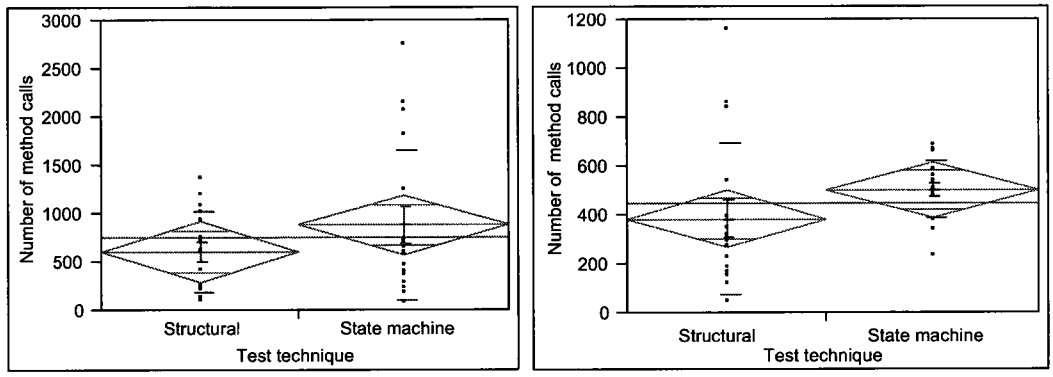
	Cruise Control			Elevator		
	|Fs∩Fc| /|F|%	|Fs∩Fc| / |Fs|	|Fs∩Fc| / |Fc|	|Fs∩Fc| /|F|%	|Fs∩Fc| / |Fs|	|Fs∩Fc| / |Fc|
Median	28.67	0.78	0.59	38.52	0.88	0.89
Mean	26.29	0.76	0.56	39.54	0.87	0.84
Min	9.95	0.46	0.25	29.42	0.51	0.66
Max	36.13	0.96	0.68	54.08	1.18	0.97
95%	35.74	0.94	0.68	52.18	1.13	0.96
90%	31.78	0.94	0.68	49.96	1.08	0.94
75%	29.78	0.82	0.66	45.83	0.98	0.92
25%	22.32	0.75	0.49	33.08	0.76	0.72
10%	17.25	0.48	0.38	30.85	0.64	0.67
5%	15.83	0.47	0.36	29.73	0.61	0.66

Table 65: Intersection score and ratios (Sannio 2)

	Cruise Control				Elevator			
	|FsUFc| /|F|%	|F – (FsUFc)| /|F|%	|FsUFc| / |Fs|	|FsUFc| / |Fc|	|FsUFc| /|F|%	|F – (FsUFc)| /|F|%	|FsUFc| / |Fs|	|FsUFc| / |Fc|
Median	52.88	47.12	1.46	1.18	55.36	44.64	1.12	1.11
Mean	55.87	44.13	1.67	1.21	55.11	44.89	1.23	1.17
Min	41.10	21.47	1.34	1.00	37.50	30.27	0.90	1.02
Max	78.53	58.90	3.87	1.66	69.73	62.50	1.82	1.72
95%	78.27	52.62	2.45	1.64	65.02	60.60	1.75	1.48
90%	77.80	50.29	2.37	1.56	63.38	53.46	1.61	1.46
75%	55.43	48.69	1.81	1.20	61.82	49.49	1.37	1.22
25%	51.31	44.57	1.38	1.13	50.51	38.18	1.00	1.05
10%	49.71	22.20	1.36	1.02	46.54	36.62	0.98	1.03
5%	47.38	21.73	1.35	1.02	39.40	34.98	0.94	1.02

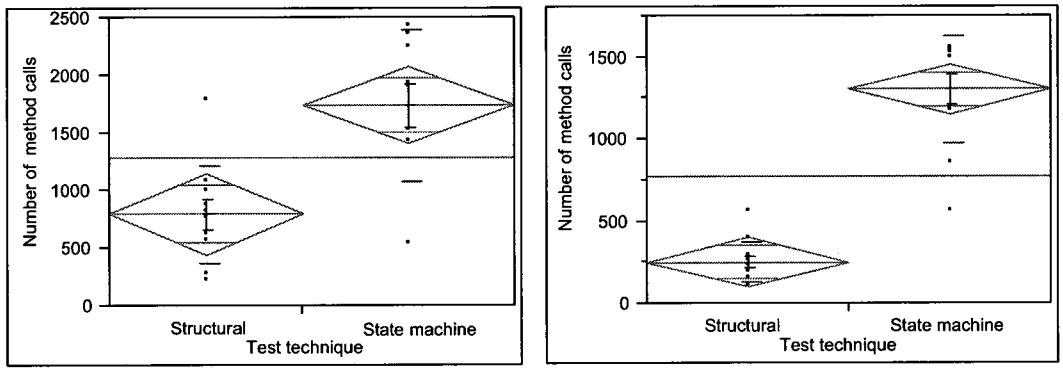
Table 66: Union and not detected faults scores and ratios (Sannio 2)

Appendix G Cost distribution and correlation with mutation score



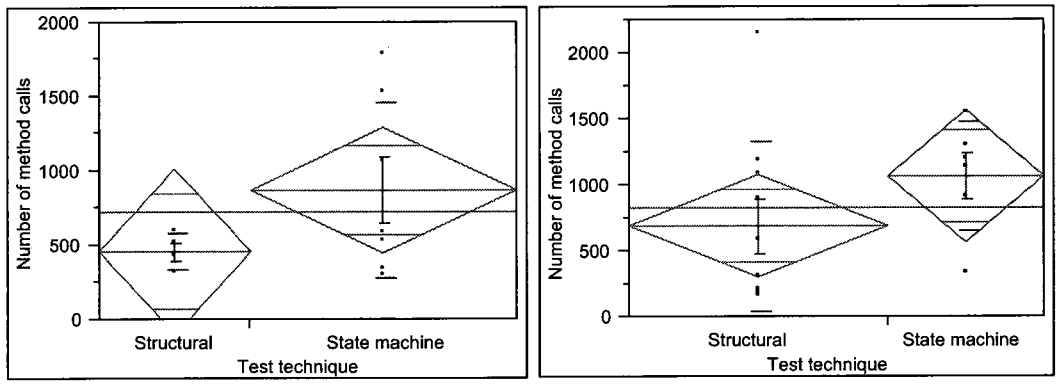
a) OrdSet b) Cruise Control

Figure 62: Cost comparison of state and structural drivers (Carleton 1)



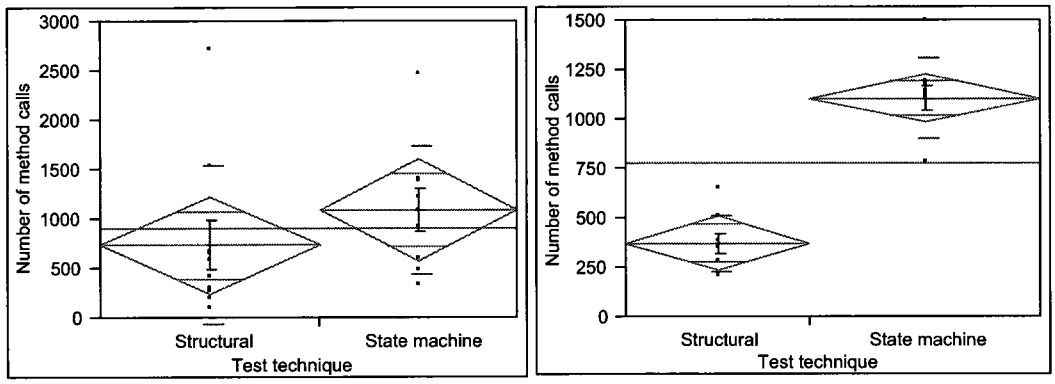
a) OrdSet b) Cruise Control

Figure 63: Cost comparison of state and structural drivers (Sannio 1)



a) Elevator b) Cruise Control

Figure 64: Cost comparison of state and structural drivers (Carleton 2)



a) Elevator b) Cruise Control

Figure 65: Cost comparison of state and structural drivers (Sannio 2)

Appendix H Node coverage and test technique combined impact on mutation scores

Experiment	System	RSquare	Factor	Parameter estimate	Sum of Squares	F Ratio	Prob > F
Carleton 1	**Cruise Control**	0.49	Test technique	0.9	23.72	0.89	**<.0001**
			Node coverage	0.88	566.67	21.39	0.3518
	OrdSet	0.77	Test technique	0.67	15.48	0.18	0.6678
			Node coverage	1.09	8780.85	106.38	**<.0001**
Sannio 1	**Cruise Control**	0.42	Test technique	2.88	187.82	1.89	0.1831
			Node coverage	1.46	1007.21	10.16	**0.0044**
	OrdSet	0.56	Test technique	-3.76	294.06	4.01	0.0589
			Node coverage	1.46	1840.39	25.11	**<.0001**
Carleton 2	**Cruise Control**	0.74	Test technique	3.35	133.58	1.99	0.1796
			Node coverage	1.42	1378.63	20.59	**0.0005**
	Elevator	0.66	Test technique	-0.2	0.46	0.008	0.9307
			Node coverage	0.8	1048.63	18	**0.0022**
Sannio 2	**Cruise Control**	0.43	Test technique	2.72	84.3	1.06	**0.3169**
			Node coverage	1.28	303.53	3.84	0.0676
	Elevator	0.88	Test technique	-0.53	5.66	0.24	0.628
			Node coverage	0.78	2948.83	126.79	**<.0001**

Table 67: Model regression results of node coverage and test technique impact on mutation scores

Appendix I Mutation scores per mutation operator

Mutation scores descriptive statistics in:			Median	Mean	95%	90%	75%	25%	10%	5%	Min	Max
	JDC (1)	Fs	100	100	100	100	100	100	100	100	100	100
		Fs-Fc	0	0	0	0	0	0	0	0	0	0
	EAM (5)	Fs	0	9.33	20	20	20	0	0	0	0	20
		Fs-Fc	0	5.6	20	20	20	0	0	0	0	20
	AORB (28)	Fs	0	0	0	0	0	0	0	0	0	0
		Fs-Fc	0	0	0	0	0	0	0	0	0	0
	AOIU (31)	Fs	28.13	28.34	28.13	31.25	29.07	28.13	28.13	28.13	28.13	28.13
		Fs-Fc	3.13	6.56	0	18.75	18.75	18.75	15.63	0	0	0
	AOIS (148)	Fs	23.61	23.57	21.53	25.7	25.21	24.72	24.31	22.92	22.22	22.01
		Fs-Fc	11.11	10.94	4.17	18.06	18.06	14.79	13.02	8.33	6.94	5.56
	ROR (79)	Fs	29.11	29.11	29.11	29.11	29.11	29.11	29.11	29.11	29.11	29.11
		Fs-Fc	3.8	4.81	0	13.92	13.92	10.51	6.33	1.27	0	0
	LOI (49)	Fs	52.94	52.94	52.94	52.94	52.94	52.94	52.94	52.94	52.94	52.94
		Fs-Fc	6.86	11.93	0	31.37	31.37	25.88	17.65	3.92	3.53	0
	ASRS (12)	Fs	0	0	0	0	0	0	0	0	0	0
		Fs-Fc	0	0	0	0	0	0	0	0	0	0

Table 68: Mutation scores per mutation operator statistics for Cruise Control (Carleton 1)

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

Mutation scores descriptive statistics in:			Median	Mean	95%	90%	75%	25%	10%	5%	Min	Max
	JDC (1)	Fs	100	100	100	100	100	100	100	100	100	100
		Fs-Fc	0	0	0	0	0	0	0	0	0	0
	EAM (5)	Fs	0	0	0	0	0	0	0	0	0	0
		Fs-Fc	0	0	0	0	0	0	0	0	0	0
	AORB (28)	Fs	0	0	0	0	0	0	0	0	0	0
		Fs-Fc	0	0	0	0	0	0	0	0	0	0
	AOIU (31)	Fs	38.71	35.75	40.16	38.71	38.71	36.29	29.03	24.67	19.35	41.94
		Fs-Fc	5.71	5.11	6.27	5.71	5.71	5.15	3.47	2.69	1.74	6.95
	AOIS (148)	Fs	42.57	39.92	44.9	44.52	43.41	38.51	29.39	29.05	29.05	45.27
		Fs-Fc	15.98	14.54	16.99	16.7	16.31	13.79	9.99	9.6	9.15	17.31
	ROR (79)	Fs	41.77	40.29	42.34	41.77	41.77	40.51	36.71	35	32.91	43.04
		Fs-Fc	8.47	8.01	8.87	8.47	8.47	7.94	6.62	6.27	5.84	9.35
	LOI (49)	Fs	57.14	55.44	57.14	57.14	57.14	57.14	55.3	47.76	38.78	57.14
		Fs-Fc	10.83	10.48	10.83	10.83	10.83	10.83	10.27	8.86	7.22	10.83
	ASRS (12)	Fs	66.67	50	66.67	66.67	66.67	50	0	0	0	66.67
		Fs-Fc	5.13	3.85	5.13	5.13	5.13	3.85	0	0	0	5.13

Table 69: Mutation scores per mutation operator statistics for Cruise Control (Sannio 1)

Mutation scores descriptive statistics in:			Median	Mean	95%	90%	75%	25%	10%	5%	Min	Max
	JDC (1)	Fs	100	83.33	100	100	100	100	50	25	0	100
		Fs-Fc	0	0	0	0	0	0	0	0	0	0
	EAM (5)	Fs	0	0	0	0	0	0	0	0	0	0
		Fs-Fc	0	0	0	0	0	0	0	0	0	0
	AORB (28)	Fs	0	0	0	0	0	0	0	0	0	0
		Fs-Fc	0	0	0	0	0	0	0	0	0	0
	AOIU (31)	Fs	30.65	31.18	38.71	38.71	37.10	26.62	24.19	23.38	22.58	38.71
		Fs-Fc	3.23	3.90	9.68	6.45	6.45	3.23	0	0	0	12.9
	AOIS (148)	Fs	36.49	33.78	42.40	42.23	41.55	29.39	22.635	19.76	16.89	42.57
		Fs-Fc	12.16	11.40	21.62	19.59	16.89	5.41	2.70	1.96	0	27.7
	ROR (79)	Fs	36.08	34.81	40.51	40.51	39.56	35.44	27.85	24.05	20.25	40.51
		Fs-Fc	6.33	6.06	14.11	12.66	7.59	1.27	0	0	0	21.52
	LOI (49)	Fs	54.08	52.38	57.14	57.14	57.14	51.02	45.92	43.37	40.82	66.67
		Fs-Fc	10.2	8.47	22.45	16.33	10.2	4.08	0	0	0	24.49
	ASRS (12)	Fs	33.34	33.34	66.67	66.67	66.67	0	0	0	0	66.67
		Fs-Fc	0	0	0	0	0	0	0	0	0	0

Table 70: Mutation scores per mutation operator statistics for Cruise Control (Carleton 2)

Mutation scores descriptive statistics in:			Median	Mean	95%	90%	75%	25%	10%	5%	Min	Max
	JDC (1)	Fs	100	100	100	100	100	100	100	100	100	100
		Fs-Fc	0	0	0	0	0	0	0	0	0	0
	EAM (5)	Fs	0	0	0	0	0	0	0	0	0	0
		Fs-Fc	0	0	0	0	0	0	0	0	0	0
	AORB (28)	Fs	0	0.32	1.79	0	0	0	0	0	0	3.57
		Fs-Fc	0	0.04	0	0	0	0	0	0	0	3.57
	AOIU (31)	Fs	38.71	35.78	40.33	38.71	38.71	33.87	29.03	25.81	22.58	41.94
		Fs-Fc	6.45	6.52	16.13	12.90	9.68	0	0	0	0	19.35
	AOIS (148)	Fs	42.57	39.13	44.60	43.92	43.24	39.87	28.38	23.65	18.92	45.27
		Fs-Fc	12.16	13.18	28.38	21.28	16.89	8.11	4.73	3.38	1.35	30.41
	ROR (79)	Fs	40.51	38.90	41.14	40.51	40.51	38.61	36.71	32.91	29.11	41.77
		Fs-Fc	5.06	6.93	21.52	17.72	7.59	3.80	1.27	1.27	1.26	22.78
	LOI (49)	Fs	57.14	54.91	57.14	57.14	57.14	57.14	55.10	44.90	34.69	66.67
		Fs-Fc	12.24	11.34	24.49	23.06	12.24	4.08	0.00	0.00	0	24.49
	ASRS (12)	Fs	66.67	48.49	66.67	66.67	66.67	33.34	0	0	0	66.67
		Fs-Fc	0	0	0	0	0	0	0	0	0	0

Table 71: Mutation scores per mutation operator statistics for Cruise Control (Sannio 2)

Mutation scores descriptive statistics in:			Median	Mean	95%	90%	75%	25%	10%	5%	Min	Max
	JSI (5)	Fs	40	42.86	40	60	54	48	40	40	40	40
		Fs-Ec	0	1.79	0	20	20	0	0	0	0	0
	EAM (4)	Fs	75	75	75	75	75	75	75	75	75	75
		Fs-Ec	0	9.38	0	75	75	75	0	0	0	0
	AORB (90)	Fs	75.56	69.21	38.89	91.11	88.78	86.44	81.67	57.78	39.56	39.22
		Fs-Ec	11.11	13.87	0	57.78	48.06	25.56	18.89	3.33	0	0
	AORS (8)	Fs	75	75	50	87.5	87.5	87.5	87.5	68.75	57.5	53.75
		Fs-Ec	0	8.71	0	37.5	37.5	25	12.5	0	0	0
	AOIU (48)	Fs	85.42	75.89	50	91.67	90.42	89.17	86.46	65.63	51.25	50.62
		Fs-Ec	9.38	9.97	0	41.67	37.5	14.58	12.5	2.08	0	0
	AOIS (297)	Fs	58.59	55.96	38.72	65.32	65.12	64.92	63.64	50.92	41.95	40.34
		Fs-Ec	6.73	9.39	0	36.7	31.4	17.51	13.22	1.68	0.34	0
	AODU (4)	Fs	75	82.14	75	100	100	100	87.5	75	75	75
		Fs-Ec	0	0.89	0	25	0	0	0	0	0	0
	ROR (47)	Fs	65.96	62.61	42.55	78.72	76.81	74.89	71.28	54.26	46.38	44.47
		Fs-Ec	6.38	6.08	0	21.28	14.89	13.83	10.64	0	0	0
	COR (6)	Fs	100	97.62	83.33	100	100	100	100	100	93.33	88.33
		Fs-Ec	0	9.52	0	33.33	33.33	33.33	16.67	0	0	0
	COD (1)	Fs	100	100	100	100	100	100	100	100	100	100
		Fs-Ec	0	0	0	0	0	0	0	0	0	0
	COI (4)	Fs	100	100	100	100	100	100	100	100	100	100
		Fs-Ec	12.5	18.75	0	50	50	50	31.25	0	0	0
	LOI (107)	Fs	85.05	76.5	54.21	88.79	87.95	87.1	85.52	68.23	57.01	55.61
		Fs-Ec	7.48	9.76	0	43.93	39.72	18.22	12.15	3.5	0	0

Table 72: Mutation scores per mutation operator statistics for OrdSet (Carleton 1)

Mutation scores descriptive statistics in:			Median	Mean	95%	90%	75%	25%	10%	5%	Min	Max
	JSI (5)	Fs	40	44.62	60	60	60	40	40	32	20	60
		Fs-Fc	1.82	5.59	14.55	14.55	14.55	1.82	1.82	1.09	0	14.55
	EAM (4)	Fs	75	63.46	75	75	75	75	25	25	25	75
		Fs-Fc	18.18	14.51	18.18	18.18	18.18	18.18	2.27	2.27	2.27	18.18
	AORB (90)	Fs	86.67	90	81.11	33.33	86.67	40	86.67	94.44	95.56	65.56
		Fs-Fc	23.54	21.25	27.23	26.44	24.34	23.23	9.23	6.46	5.56	28.38
	AORS (8)	Fs	75	73.08	87.5	85	75	62.5	62.5	62.5	62.5	87.5
		Fs-Fc	4.55	5.16	6.82	6.82	6.82	4.55	4.55	4.09	3.41	6.82
	AOIU (48)	Fs	85.42	79.49	92.5	91.67	91.67	81.25	54.17	51.67	47.92	93.75
		Fs-Fc	14.58	13.18	16.74	16.57	14.96	13.45	7.58	7.42	7.2	16.86
	AOIS (297)	Fs	66.67	61.67	74.55	73.74	72.73	60.94	38.39	36.9	36.7	75.76
		Fs-Fc	14.45	15.46	13.41	4.5	14.45	4.13	13.9	14.85	16.35	6.7
	AODU (4)	Fs	100	92.31	100	100	100	75	75	75	75	100
		Fs-Fc	4.55	3.5	4.55	4.55	4.55	4.55	0	0	0	4.55
	ROR (47)	Fs	78.72	74.8	85.11	85.11	85.11	76.6	51.91	51.06	51.06	85.11
		Fs-Fc	13.93	13.61	15.67	15.67	15.67	13.35	10.14	9.9	9.67	15.67
	COR (6)	Fs	83.33	80.77	100	100	100	66.67	66.67	66.67	66.67	100
		Fs-Fc	9.09	9.32	16.67	16.67	16.67	4.55	4.55	4.55	4.55	16.67
	COD (1)	Fs	100	100	100	100	100	100	100	100	100	100
		Fs-Fc	9.09	9.09	9.09	9.09	9.09	9.09	9.09	9.09	9.09	9.09
	COI (4)	Fs	75	80.77	100	100	100	75	50	50	50	100
		Fs-Fc	2.27	1.75	2.27	2.27	2.27	2.27	0	0	0	2.27
	LOI (107)	Fs	85.05	80.16	92.9	92.33	90.65	78.5	59.62	56.07	53.27	93.46
		Fs-Fc	12.66	12.24	15.19	14.9	14.44	11.89	7.65	6.81	6.46	15.55

Table 73: Mutation scores per mutation operator statistics for OrdSet (Sannio 1)

Mutation scores descriptive statistics in:			Median	Mean	95%	90%	75%	25%	10%	5%	Min	Max
	JSI (22)	Fs	6.82	10.23	24.09	20.91	14.78	4.55	3.19	1.59	0	27.27
		Fs-Fc	0	2.96	13.64	9.09	4.55	0	0	0	0	25
	JSD (2)	Fs	0	0	0	0	0	0	0	0	0	0
		Fs-Fc	0	0	0	0	0	0	0	0	0	0
	JID (6)	Fs	50	50	50	50	50	50	50	50	50	50
		Fs-Fc	0	0	0	0	0	0	0	0	0	0
	JDC (1)	Fs	100	100	100	100	100	100	100	100	100	100
		Fs-Fc	0	0	0	0	0	0	0	0	0	0
	EAM (56)	Fs	8.93	10.49	20.71	18.21	16.07	3.57	3.57	3.57	3.57	23.21
		Fs-Fc	1.79	4.95	15.63	14.29	7.14	0.45	0	0	0	23.21
	AORB (76)	Fs	2.63	5.43	19.8	11.97	5.26	0	0	0	0	27.63
		Fs-Fc	0	3.27	21.38	7.89	3.95	0	0	0	0	26.32
	AORS (6)	Fs	91.67	87.5	100	100	100	79.17	66.67	66.67	66.67	100
		Fs-Fc	0	5.3	33.33	16.67	0	0	0	0	0	33.33
	AOIU (88)	Fs	36.94	40.91	62.79	52.84	42.62	34.09	31.71	28.92	26.14	72.73
		Fs-Fc	4.55	7.06	25.85	12.5	7.95	3.41	1.7	1.14	1.14	32.95
	AOIS (556)	Fs	43.17	48.09	71.85	66.19	56.48	36.74	34.01	32.57	31.12	77.52
		Fs-Fc	12.41	15.01	35.79	28.6	19.02	8.45	5.49	3.78	1.62	45.5
	AODU (5)	Fs	0	0	0	0	0	0	0	0	0	0
		Fs-Fc	0	0	0	0	0	0	0	0	0	0
	ROR (107)	Fs	18.23	23.6	44.62	39.72	34.11	14.96	10.19	7.9	5.61	49.53
		Fs-Fc	2.8	8.86	30.37	25.7	15.42	0.93	0	0	0	44.86
	COR (28)	Fs	35.71	37.95	60.36	52.86	38.39	34.82	26.79	20.54	14.29	67.86
		Fs-Fc	10.71	13.37	33.93	25	17.86	7.14	0	0	0	42.86
	COI (18)	Fs	27.78	34.03	66.39	60.56	43.06	25	13.34	9.45	5.56	72.22
		Fs-Fc	5.56	11.95	47.22	36.11	22.22	0	0	0	0	66.67
	LOI (192)	Fs	42.19	47.27	64.09	60.99	55.6	39.32	37.6	36.51	35.42	67.19
		Fs-Fc	8.33	9.73	23.7	20.57	12.5	4.69	3.39	1.56	1.04	32.81
	ASRS (4)	Fs	0	9.38	48.75	22.5	0	0	0	0	0	75
		Fs-Fc	0	9.09	75	25	0	0	0	0	0	75
	JTD (4)	Fs	25	31.25	50	50	50	25	17.5	8.75	0	50
		Fs-Fc	0	3.41	25	25	0	0	0	0	0	25
	JTI (5)	Fs	20	25	40	40	40	20	14	7	0	40
		Fs-Fc	0	2.73	20	20	0	0	0	0	0	20

Table 74: Mutation scores per mutation operator statistics for Elevator (Carleton 2)

Mutation scores descriptive statistics in:			Median	Mean	95%	90%	75%	25%	10%	5%	Min	Max
	JSI (22)	Fs	4.55	9.55	23.18	19.09	14.77	4.55	4.55	4.55	4.55	27.27
		Fs-Fc	0	3.45	13.86	13.64	4.55	0	0	0	0	22.73
	JSD (2)	Fs	0	0	0	0	0	0	0	0	0	0
		Fs-Fc	0	0	0	0	0	0	0	0	0	0
	JID (6)	Fs	50	50	50	50	50	50	50	50	50	50
		Fs-Fc	0	0	0	0	0	0	0	0	0	0
	JDC (1)	Fs	100	100	100	100	100	100	100	100	100	100
		Fs-Fc	0	0	0	0	0	0	0	0	0	0
	EAM (56)	Fs	6.25	9.47	20.8	18.4	14.29	5.36	3.39	2.59	1.79	23.21
		Fs-Fc	1.79	4.52	17.86	14.29	5.36	0	0	0	0	21.43
	AORB (76)	Fs	2.63	5.26	19.34	11.05	5.26	0	0	0	0	27.63
		Fs-Fc	0	3.26	19.8	10	3.95	0	0	0	0	27.63
	AORS (6)	Fs	83.33	86.67	100	100	95.83	83.33	81.66	74.17	66.67	100
		Fs-Fc	0	6	33.33	16.67	16.67	0	0	0	0	33.33
	AOIU (88)	Fs	38.64	43.3	61.99	51.25	46.31	37.5	33.98	33.46	32.95	72.73
		Fs-Fc	6.82	8.83	26.14	22.84	11.65	1.99	1.14	1.14	1.13	44.32
	AOIS (556)	Fs	42.54	46.01	71.29	65.06	55.22	37.73	30.77	26.8	22.84	77.52
		Fs-Fc	8.9	14.51	40.61	36.47	20.19	4.63	2.86	1.62	0.9	56.29
	AODU (5)	Fs	0	2	11	2	0	0	0	0	0	20
		Fs-Fc	0	0	0	0	0	0	0	0	0	0
	ROR (107)	Fs	19.16	22.9	44.49	39.44	32.25	14.48	11.22	7.01	2.8	49.53
		Fs-Fc	4.21	9.93	35.51	30.09	16.82	0.93	0	0	0	46.73
	COR (28)	Fs	35.71	39.64	58.22	48.57	41.07	35.71	34.64	29.82	25	67.86
		Fs-Fc	14.29	18.32	42.86	32.5	25	10.71	6.79	0	0	60.71
	COI (18)	Fs	27.78	35	62.22	52.22	44.45	27.78	26.11	18.61	11.11	72.22
		Fs-Fc	5.56	12.72	50	33.33	16.67	0	0	0	0	61.11
	LOI (192)	Fs	43.49	46.2	63.2	59.22	53.52	38.93	36.82	33.78	30.73	67.19
		Fs-Fc	5.73	9.96	28.67	26.15	15.1	3.13	1.56	1.56	0.52	39.06
	ASRS (4)	Fs	0	7.5	41.25	7.5	0	0	0	0	0	75
		Fs-Fc	0	5.75	48.75	0	0	0	0	0	0	75
	JTD (4)	Fs	25	27.5	50	50	43.75	25	0	0	0	50
		Fs-Fc	0	9	25	25	25	0	0	0	0	50
	JTI (5)	Fs	20	22	40	40	35	20	0	0	0	40
		Fs-Fc	0	7.2	20	20	20	0	0	0	0	40

Table 75: Mutation scores per mutation operator statistics for Elevator (Sannio 2)

Appendix J OrdSet state machine guard conditions

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

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

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

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

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

Appendix K Contracts

K.1 Cruise Control's Contracts

K.1.1 CruiseControl class

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

K.1.2 SpeedControl class

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

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

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

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

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

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

K.1.3 CarSimulator class

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

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

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

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

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

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

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

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

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

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

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

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

K.1.4 Controller class

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

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

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

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

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

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

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

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

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

K.2 OrdSet's Contracts

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

context OrdSet::defSetSize(n: int): int
pre: n >= 0
post: (n < min_set_size) implies (result = min_set_size)
```

```
          and  (n >= max_set_size) implies (result = max_set_size)
          and  ((n > min_set_size and n < max_set_size) implies
        (result.mod(min_set_size) = 0  and result <
          n+min_set_size))

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

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

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

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

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

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

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

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

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

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

context OrdSet::isEmpty(): boolean
pre: none
post: result = _set->isEmpty()

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

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

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

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

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

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

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

context OrdSet::make_a_free_slot(n: int): int
pre: none
post: result >= 0
      and  result <= _last@pre+1)
      and  Sequence{0..result-1}->forAll(i|self.elementAt(i) =
            self@pre.elementAt(i)
      and  Sequence{result+1.._last}->forAll(i|self.elementAt(i) =
            self@pre.elementAt(i-1)
```

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

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

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

K.3 Elevator's Contracts

K.3.1 Elevator class

```
context Elevator
inv: elevatorID >= 0
and Sequence{0..ElevatorGroup.numElevators-1)->
forall (x:int| Elevator.selectElevator(x).getElevatorID() = x)
 and
self.state = #IDLE implies (Sequence{0..ElevatorGroup.numFloors-1)
-> forall(x:int| self.stops[x]=false) and self.doorOpen
and not self.motorMoving)
 and
self.state = #PREPARE implies (Sequence{0..ElevatorGroup.numFloors
-1) -> exists(x:int| self.stops[x]=true) and not self.doorOpen and
not self.motorMoving)
 and
self.state = #MOVING implies (Sequence{0.. ElevatorGroup.numFloors
-1) -> exists(x:int| self.stops[x]=true) and not self.doorOpen and
 self.motorMoving)
 and
self.state = #FINDNEXT implies (self.doorOpen and not
 self.motorMoving)

context Elevator::selectElevator(elevatorID: int): Elevator
pre: elevatorID >= 0 and elevatorID < ElevatorGroup.numElevators
post: result = Elevator.allInstances -> select(e: Elevator |
e.elevatorID = elevatorID)

context Elevator::getBestElevator(floorID: int): Elevator
pre: floorID >= 0 and floorID < ElevatorGroup.numFloors
post: (result.state = #IDLE and
Sequence{0..ElevatorGroup.numElevators -1)->
select(i|Elevator.selectElevator(i).state=#IDLE)-> forAll(i:int|
(abs(floorID -Elevator.selectElevator(i).getFloor()) > (abs(floorID
 - result.getFloor())))
or
(Sequence{0..ElevatorGroup.numElevators-1)-> forAll(i|
```

```
Elevator.selectElevator(i).state != #IDLE) and ((result.getFloor()-
floorID*self.direction <=0) and
Sequence{0..ElevatorGroup.numElevators-1)-> select(i|
Elevator.selectElevator(i).getFloor()-floorID*
Elevator.selectElevator(i).direction <=0)->forAll(i|
abs(Elevator.selectElevator(i).getFloor()-floorID) >=
abs(result.getFloor()-floorID))))
or
(Sequence{0..ElevatorGroup.numElevators-1)-> forAll(i|
Elevator.selectElevator(i).state != #IDLE) and
Sequence{0..ElevatorGroup.numElevators-1)-> forAll(i|
 Elevator.selectElevator(i).getFloor()-floorID*
Elevator.selectElevator(i).direction > 0))

context Elevator::notifyNewFloor(newFloor: Floor): boolean
pre: none
post: result = self.stops[newFloor.floorID] and self.currentFloor =
 newFloor

context Elevator::moveElevator()
pre: none
post: none

context Elevator::stopElevator()
pre: self.stops[self.currentFloor]
post: self.doorOpen and not self.motorMoving and not
 self.stops[self.currentFloor] and state = #FINDNEXT

context Elevator::openDoor()
pre: not self.doorOpen
post: self.doorOpen

context Elevator::closeDoor()
pre: self.doorOpen
post: not self.doorOpen

context Elevator::addStop (floorID: int, stopState:boolean)
pre: stopState xor self.stops[floorID]
post: self.stops[floorID] = stopState

context Elevator::getDirection ()
pre: none
post: result = self.direction

context Elevator::getState(): int
pre: none
post: result = self.state

context Elevator::getFloor(): Floor
pre: none
post: result = self.currentFloor

context Elevator::getElevatorID(): int
pre: none
post: result = self.elevatorID

context Elevator::getDoorOpen(): boolean
```

```
pre: none
post: result = self.doorOpen

context Elevator::getMotorMoving(): boolean
pre: none
post: result = self.motorMoving
```

K.3.2 ElevatorControl class

```
context ElevatorControl::stopElevator()
pre: self.myElevator.motorMoving and not self.myElevator.doorOpen
post: not self.myElevator.motorMoving and self.myElevator.doorOpen
 and self.myElevator.state = #FINDNEXT

context ElevatorControl::openDoor()
pre: not self.myElevator.doorOpen
post: self.myElevator.doorOpen

context ElevatorControl::closeDoor()
pre: self.myElevator.doorOpen
post: not self.myElevator.doorOpen

context ElevatorControl::requestStop(floor: int)
pre: none
post: self.myElevator.stops[floor]

context ElevatorControl::getElevator()
pre: none
post: result = self.myElevator

context ElevatorControl::motorMoveDown()
pre: none
post: none

context ElevatorControl::motorMoveUp()
pre: none
post: none

context ElevatorControl::motorStop()
pre: none
post: none
```

K.3.3 ElevatorInterface class

```
context ElevatorInterface::getFromList(ec: ElevatorControl):
ElevatorInterface
pre: none
post: result.elevatorID = ec.myElevator.elevatorID

context ElevatorInterface::requestStop(floor: int)
pre: floor >= 0 and floor < ElevatorGroup.numFloors
post: (Elevator.allInstances -> select(e|e.elevatorID =
self.elevatorID)).stops[floor]=true

context ElevatorInterface::getElevatorID(): int
```

```
pre: none
post: result = self.elevatorID

context ElevatorInterface::motorMoveDown()
pre: none
post: none

context ElevatorInterface::motorMoveUp()
pre: none
post: none

context ElevatorInterface::motorStop()
pre: none
post: none
```

K.3.4 Floor class

```
context Floor
inv: floorID >= 0 and floorID < ElevatorGroup.numFloors and
allFloors -> notEmpty

context Floor::getNoFloors(): int
pre: none
post: result = ElevatorGroup.numFloors

context Floor::selectFloor(floorID: int): Floor
pre: floorID >= 0 and floorID < ElevatorGroup.numFloors
post: result.floorID = floorID

context Floor::requestUp(): Elevator
pre: self.floorID < ElevatorGroup.numFloors - 1
post: result = Elevator.getBestElevator(self.floorID) and
result.stops[self.floorID] = true

context Floor::requestDown(): Elevator
pre: self.floorID > 0
post: result = Elevator.getBestElevator(self.floorID) and
result.stops[self.floorID] = true

context Floor::requestUpMade(): boolean
pre: none
post: result = self.upButtonPressed

context Floor::requestDownMade(): boolean
pre: none
post: result = self.downButtonPressed

context Floor::requestUpServiced()
pre: upButtonPressed
post: not upButtonPressed

context Floor::requestDownServiced()
pre: downButtonPressed
post: not downButtonPressed

context Floor::getFloorID(): int
```

```
pre: none
post: result = self.floorID

context Floor::getSensor(): ArrivalSensor
pre: none
post: result = self.arrivalSensor

context Floor::removeFloors()
pre: none
post: allFloors = null
```

K.3.5 ArrivalSensor class

```
context ArrivalSensor::stopAtThisFloor(elevatorID: int): boolean
pre: none
post: result = (Elevator.allInstances->select(e|e.elevatorID =
elevatorID)).stops[self.floor]

context ArrivalSensor::getTheFloor(): Floor
pre: none
post: result = self.floor
```

K.3.6 FloorControl class

```
context FloorControl::requestUp(floorID: int): Elevator
pre: floorID < ElevatorGroup.numFloors - 1
post: result = Elevator.getBestElevator(floorID) and
result.stops[floorID] = true

context FloorControl::requestDown(floorID: int): Elevator
pre: floorID > 0
post: result = Elevator.getBestElevator(floorID) and
result.stops[floorID] = true

context FloorControl::stopAtThisFloor(elevatorID: int, floorID:
int)
pre: none
post: result = (Elevator.allInstances->select(e|e.elevatorID =
elevatorID)).stops[floorID]
```

K.3.7 FloorInterface class

```
context FloorInterface::requestUp(floorID: int): Elevator
pre: floorID >= 0 and floorID < ElevatorGroup.numFloors - 1
post: result = Elevator.getBestElevator(floorID) and
result.stops[floorID] = true

context FloorInterface::requestDown(floorID: int): Elevator
pre: floorID > 0 and floorID < ElevatorGroup.numFloors - 1
post: result = Elevator.getBestElevator(floorID) and
result.stops[floorID] = true

context FloorInterface::stopAtThisFloor(elevatorID: int, floorID:
int)
```

```
pre: none
post: result = (Elevator.allInstances->select(e|e.elevatorID =
elevatorID)).stops[floorID]

context FloorInterface::getFloor(): Floor
pre: none
post: self.sensor.floor
```

K.3.8 ElevatorGroup class

```
context ElevatorGroup::getGroup(el: int, fl: int): ElevatorGroup
pre: none
post: self.numFloors = fl and self.numElevators = el

context ElevatorGroup::startThread(threadNum: int)
pre: none
post: none

context ElevatorGroup::stopGroup()
pre: none
post: none

context ElevatorGroup::startGroup()
pre: none
post: none

context ElevatorGroup::motorMoving(elevatorID: int, direction: int,
currentFloor: int)
pre: none
post: none

context ElevatorGroup::elevatorDisplay(eid: int, message: String)
pre: none
post: none

context ElevatorGroup::getFloorInterface(floorID: int):
FloorInterface
pre: none
post: result = fli[floorID]

context ElevatorGroup::getElevatorInterface(elevatorID: int):
ElevatorInterface
pre: none
post: result = ebi[floorID]
```

Appendix L Survey questionnaires

The questionnaires that were distributed to participants are provided below. Questionns varied across questionnaires depending on whether the lab was the first one and whether the test technique used was the state test technique or the structural code coverage test technique.

Survey Questionnaire

Levels of agreement:

1 – Strongly agree 2 – Agree 3 – Not certain 4 – Disagree 5 – Strongly disagree

Questions about the task	**1**	**2**	**3**	**4**	**5**
1. It took me a lot of time to understand the system before I could perform the task.	☐	☐	☐	☐	☐
2. I did not have any problems to understand what the system is doing.	☐	☐	☐	☐	☐
3. I needed more time to complete the task.	☐	☐	☐	☐	☐
4. The lab instructions were clear and easy to follow.	☐	☐	☐	☐	☐
Questions about testing	**1**	**2**	**3**	**4**	**5**
5. Identifying test cases based on my understanding of the code was easy.	☐	☐	☐	☐	☐
6. I wished I had access to more information on the system (other than code) to help me identify test cases.	☐	☐	☐	☐	☐
7. I wished I had access to more information on the system (other than code) to help me identify oracles.	☐	☐	☐	☐	☐
Questions about tools	**1**	**2**	**3**	**4**	**5**
9. I am an expert in using Eclipse.	☐	☐	☐	☐	☐
10. I have often used Eclipse before the lab.	☐	☐	☐	☐	☐
11. The instrumented code was helpful to identify test cases to improve structural coverage.	☐	☐	☐	☐	☐

8. How much time did you spend on understanding the system and its code during this lab?

A. <25% B. >=25% and <50% C. >=50% and <75% D: >=75%

Table 76: Survey Questionnaire for lab 1 (structural testing)

Survey Questionnaire

Levels of agreement:

1 – Strongly agree 2 – Agree 3 – Not certain 4 – Disagree 5 – Strongly disagree

Questions about the task	**1**	**2**	**3**	**4**	**5**
1. It took me a lot of time to understand the system before I could perform the task.	☐	☐	☐	☐	☐
2. I needed more time to complete the task.	☐	☐	☐	☐	☐
3. The lab instructions were clear and easy to follow.	☐	☐	☐	☐	☐
Questions about testing	**1**	**2**	**3**	**4**	**5**
4. The model artifacts helped me identify test cases.	☐	☐	☐	☐	☐
5. Without the model I would have been able to easily identify equivalent test cases.	☐	☐	☐	☐	☐
6. State invariants and contracts helped me easily identify and implement oracles.	☐	☐	☐	☐	☐
7. I could have identified effective oracles without state invariants and contracts.	☐	☐	☐	☐	☐
Questions about tools	**1**	**2**	**3**	**4**	**5**
9. I am an expert in using Eclipse.	☐	☐	☐	☐	☐
10. I have often used Eclipse before the lab.	☐	☐	☐	☐	☐

8. How much time did you spend on understanding the system and its model during this lab?

A. <25% B. >=25% and <50% C. >=50% and <75% D: >=75%

Table 77: Survey Questionnaire for lab 1 (state testing)

Survey Questionnaire

Levels of agreement:

1 – Strongly agree 2 – Agree 3 – Not certain 4 – Disagree 5 – Strongly disagree

Questions about the task	**1**	**2**	**3**	**4**	**5**
1. It took me a lot of time to understand the system before I could perform the task.	☐	☐	☐	☐	☐
2. I needed more time to complete the task.	☐	☐	☐	☐	☐
3. The lab instructions were clear and easy to follow.	☐	☐	☐	☐	☐
4. The system tested in previous lab was easier to understand.	☐	☐	☐	☐	☐
5. The system tested in previous lab was more complex.	☐	☐	☐	☐	☐
6. It was clearly helpful to have the model to better understand the system.	☐	☐	☐	☐	☐

Questions about testing	**1**	**2**	**3**	**4**	**5**
7. The model of the system helped me identify test cases easily.	☐	☐	☐	☐	☐
8. Without the model I would have been able to easily identify the same test cases.	☐	☐	☐	☐	☐
9. State invariants and contracts helped me easily identify and implement oracles.	☐	☐	☐	☐	☐
10. I could have identified equivalent oracles without state invariants and contracts.	☐	☐	☐	☐	☐
11. If I had been given the model (state diagram, contracts and state invariants …), in addition to the code in previous lab, it would have been easier to identify test cases.	☐	☐	☐	☐	☐

12. How much time did you spend on understanding the system and its model during this lab?

A. <25% B. >=25% and <50% C. >=50% and <75% D: >=75%

Questions about tools	**1**	**2**	**3**	**4**	**5**
13. I am an expert in using Eclipse.	☐	☐	☐	☐	☐
14. I have often used Eclipse before the lab.	☐	☐	☐	☐	☐

Table 78: Survey Questionnaire for lab 2 (state testing)

Survey Questionnaire

Levels of agreement:

1 – Strongly agree 2 – Agree 3 – Not certain 4 – Disagree 5 – Strongly disagree

Questions about the task	**1**	**2**	**3**	**4**	**5**
1. It took me a lot of time to understand the system before I could perform the task.	☐	☐	☐	☐	☐
2. I needed more time to complete the task.	☐	☐	☐	☐	☐
3. The lab instructions were clear and easy to follow.	☐	☐	☐	☐	☐
4. It would have been helpful to understand the system if its model was provided.	☐	☐	☐	☐	☐
5. The system tested in previous lab was easier to understand.	☐	☐	☐	☐	☐
6. The system tested in previous lab was more complex.	☐	☐	☐	☐	☐

Questions about testing	**1**	**2**	**3**	**4**	**5**
7. Identifying test cases based on my understanding of the code was easy.	☐	☐	☐	☐	☐
8. I wished I had access to the model of the system to help me identify test cases.	☐	☐	☐	☐	☐
9. I wished I had access to the model of the system to help me identify oracles.	☐	☐	☐	☐	☐
10. Identifying test cases in previous lab (model-based) was easier than identifying test cases in this lab.	☐	☐	☐	☐	☐

11. How much time did you spend on understanding the system and its code during this lab?

A. <25% B. >=25% and <50% C. >=50% and <75% D: >=75%

Questions about tools	**1**	**2**	**3**	**4**	**5**
12. I am an expert in using Eclipse.	☐	☐	☐	☐	☐
13. I have often used Eclipse before the lab.	☐	☐	☐	☐	☐
14. The instrumented code was helpful to identify test cases to improve structural coverage.	☐	☐	☐	☐	☐

Table 79: Survey Questionnaire for lab 2 (sttructural testing)

Appendix M Survey data analysis

		Cruise Control	OrdSet	p-value (t-test)	p-value (Wilcoxon)
1	Too long to understand the system	3.19	3.48	0.2593	0.2719
2	Difficulties understanding the system	2.74	2.33	0.1122	0.083
3	Not enough time to complete the task	2.83	2.17	**0.0294**	**0.0294**
4	Lab instructions clear	2.17	2.09	0.6165	0.6746
5	Improved understanding if model provided	2.27	2.67	0.355	0.346
6	System in lab 1 easier to understand	3.04	2.92	0.7279	0.6983
7	System in lab 1 more complex	3	2.78	0.5949	0.6018
8	Having the model helped understanding the system	1.8	2	0.6769	0.6184

Table 80: Comprehension and lab time availability by *Cluster* (Cruise Control & OrdSet)

		Cruise Control	Elevator	p-value (t-test)	p-value (Wilcoxon)
1	Too long to understand the system	3.11	2.02	**0.0003**	**0.0009**
2	Difficulties understanding the system	4.1	3.12	0.1424	0.1827
3	Not enough time to complete the task	2.72	1.32	**<.0001**	**<.0001**
4	Lab instructions clear	2.25	2.47	0.3474	0.1886
5	Improved understanding if model provided	3.01	1	**0.0163**	**0.0416**
6	System in lab 1 easier to understand	4.31	2.82	**0.0003**	**0.0009**
7	System in lab 1 more complex	1.41	3.61	**<.0001**	**<.0001**
8	Having the model helped understanding the system	1.77	2	0.4178	0.4262

Table 81: Comprehension and lab time availability by *Cluster* (Cruise Control & Elevator)

		Lab 1	Lab 2	p-value (t-test)	p-value (Wilcoxon)
1	Too long to understand the system	2.74	3.22	0.0192	0.0188
2	Difficulties understanding the system	3	2.4	0.0145	0.0174
3	Not enough time to complete the task	2	2.69	0.0128	0.0227
4	Lab instructions clear	2.19	2.13	0.782	0.7256

Table 82: Comprehension and lab time availability by *Lab*

		High ability	Low ability	p-value (t-test)	p-value (Wilcoxon)
1	Too long to understand the system	3.07	2.88	0.3296	0.3279
2	Difficulties understanding the system	2.78	2.67	0.6441	0.6775
3	Not enough time to complete the task	2.23	2.38	0.4964	0.4222
4	Lab instructions clear	2.18	2.3	0.4792	0.4462
5	Improved understanding if model provided	2.37	2.43	0.8922	0.9131
6	System in lab 1 easier to understand	3.04	3	0.8898	0.8848
7	System in lab 1 more complex	2.65	2.84	0.522	0.4904
8	Having the model helped understanding the system	1.94	1.84	0.737	0.6218

Table 83: Comprehension and lab time availability by *Ability*

		Carleton1	Sannio1	Carleton2	Sannio2
1	Too long to understand the system	3.39	3.27	2.97	2.24
2	Difficulties understanding the system	2.45	2.62	3.62	3.7
3	Not enough time to complete the task	2.2	2.82	1.82	2.24
4	Lab instructions clear	2.32	1.92	2.57	2.16
5	Improved understanding if model provided	2.46	2.5	2.14	2.37
6	System in lab 1 easier to understand	3.17	2.62	3.47	2.72
7	System in lab 1 more complex	2.68	3.25	2.47	2.61
8	Having the model helped understanding the system	1.86	2	1.6	2.06

Table 84: Comprehension and lab time availability by *Experiment*

		Cruise Control	OrdSet	p-value (t-test)	p-value (Wilcoxon)
1	State diagram and transition tree help determine test cases	1.62	1.89	0.3993	0.8299
2	Equivalent test cases can be identified without State diagram	3.27	3.55	0.3505	0.3306
3	State invariants and contracts help identify oracles	2.61	2.92	0.3434	0.3792
4	Effective oracles identifiable without state invariants and contracts	3.16	3.47	0.2985	0.1433
5	Identifying test cases based on code was easy	2.69	2.55	0.5465	0.4965
6	Need more system information to identify test cases	2.73	3	0.4141	0.3246
7	Need more system information to identify oracles	2.93	3	0.837	0.7639
8	Easier to identify test cases in state testing (based on two labs)	3	2.17	0.0845	0.156
9	Having state machine in previous lab (structural testing) would have helped identifying test cases	2.64	1.9	0.1915	0.0829

Table 85: Test cases and oracles identification by *Cluster* (CruiseControl & OrdSet)

		Cruise Control	Elevator	p-value (t-test)	p-value (Wilcoxon)
1	State diagram and transition tree help determine test cases	1.62	2.52	**0.0018**	**0.001**
2	Equivalent test cases can be identified without State diagram	3.27	3.43	0.4844	0.8949
3	State invariants and contracts help identify oracles	2.28	2.8	0.059	0.05
4	Effective oracles identifiable without state invariants and contracts	3.16	3.14	0.8982	0.9928
5	Identifying test cases based on code was easy	2.69	3.27	0.0598	0.0755
6	Need more system information to identify test cases	2.73	1.6	**0.001**	**0.0053**
7	Need more system information to identify oracles	2.93	2.36	0.1074	0.141
8	Easier to identify test cases in state testing (based on two labs)	3	1	**0.0002**	**0.0362**
9	Having state machine in previous lab (structural testing) would have helped identifying test cases	2.63	2.35	0.3688	0.3745

Table 86: Test cases and oracles identification by *Cluster* (CruiseControl & Elevator)

		Lab 1	Lab 2	p-value (t-test)	p-value (Wilcoxon)
1	State diagram and transition tree help determine test cases	2	1.9	0.655	0.2899
2	Equivalent test cases can be identified without state machine diagram	3.52	3.3	0.2814	0.3104
3	State invariants and contracts help identify oracles	2.59	2.37	0.3291	0.1764
4	Effective oracles identifiable without state invariants and contracts	3.14	3.27	0.4787	0.3704
5	Identifying test cases based on code was easy	2.84	2.5	0.142	0.1466
6	Need more system information to identify test cases	2.59	2.64	0.8945	0.7906
7	Need more system information to identify oracles	2.79	3	0.4968	0.4403

Table 87: Test cases and oracles identification by *Lab*

		High ability	Low ability	p-value (t-test)	p-value (Wilcoxon)
1	State diagram and transition tree help determine test cases	1.95	1.91	0.8569	0.5929
2	Equivalent test cases can be identified without state machine diagram	3.5	3.23	0.1857	0.3097
3	State invariants and contracts help identify oracles	2.4	2.5	0.711	0.833
4	Effective oracles identifiable without state invariants and contracts	3.41	2.97	**0.0187**	**0.0136**
5	Identifying test cases based on code was easy	2.81	2.57	0.2622	0.2764
6	Need more system information to identify test cases	2.53	2.68	0.6432	0.5342
7	Need more system information to identify oracles	2.65	3	0.165	0.168
8	Easier to identify test cases in state testing (based on two labs)	2.37	2.53	0.7454	0.82
9	Having state machine in previous lab (structural testing) would have helped identifying test cases	2.39	2.38	0.9907	0.8887

Table 88: Test cases and oracles identification by *Ability*

		Carleton1	Sannio1	Carleton2	Sannio2
1	State diagram and transition tree help determine test cases	1.78	1.72	2.05	2.11
2	Equivalent test cases can be identified without State diagram	4.07	3.22	3.44	3.11
3	State invariants and contracts help identify oracles	1.86	2.42	2.44	2.8
4	Effective oracles identifiable without state invariants and contracts	3.57	3.1	3	3.28
5	Identifying test cases based on code was easy	2.64	2.47	2.93	3.1
6	Need more system information to identify test cases	2.6	3.06	2.27	2.33
7	Need more system information to identify oracles	2.4	3.47	2.73	2.6
8	Easier to identify test cases in state testing (based on two labs)	2.07	2.71	3	2.82
9	Having state machine in previous lab (structural testing) would have helped identifying test cases	2	1.71	2.1	2.75

Table 89: Test cases and oracles identification by *Experiment*

		Cruise Control	Elevator	p-value (t-test)	p-value (Wilcoxon)
1	Expert in Eclipse	2.35	2.53	0.5701	0.5216
2	Used Eclipse often before lab	2.05	1.93	0.7228	0.9483
3	Instrumented code is helpful to identify test cases	1.98	1.73	0.4231	0.5082

Table 90: Experience with tools by *Cluster* (Cruise Control & Elevator)

		Cruise Control	OrdSet	p-value (t-test)	p-value (Wilcoxon)
1	Expert in Eclipse	2.35	2.92	0.7791	0.6618
2	Used Eclipse often before lab	2.05	1.93	0.5943	0.8188
3	Instrumented code is helpful to identify test cases	1.98	1.79	0.3717	0.4064

Table 91: Experience with tools by *Cluster* (Cruise Control & OrdSet)

		Lab 1	Lab 2	p-value (t-test)	p-value (Wilcoxon)
1	Expert in Eclipse	2.46	2.26	0.3234	0.2106
2	Used Eclipse often before lab	2.17	1.84	0.1309	0.1341
3	Instrumented code is helpful to identify test cases	2	1.75	0.2087	0.2278

Table 92: Experience with tools by *Lab*

		High ability	Low ability	p-value (t-test)	p-value (Wilcoxon)
1	Expert in Eclipse	2.16	2.57	**0.042**	**0.0152**
2	Used Eclipse often before lab	1.82	2.18	**0.0466**	0.0598
3	Instrumented code is helpful to identify test cases	1.88	1.89	0.9799	0.6609

Table 93: Experience with tools by *Ability*

		Carleton1	Sannio1	Carleton2	Sannio2
1	Expert in Eclipse	2.16	2.5	2.45	2.2
2	Used Eclipse often before lab	1.9	2.18	1.88	2.03
3	Instrumented code is helpful to identify test cases	1.86	1.83	1.73	2.4

Table 94: Experience with tools by *Experiment*

		Cruise Control				OrdSet				Elevator			
		<25	25-50	50-75	>75	<25	25-50	50-75	>75	<25	25-50	50-75	>75
1	Lab time to understand system and model	43	48	9	0	44	39	11	6	14	27	36	23
2	Lab time to write state driver	0	33	61	6	0	22	44	34	22	34	31	13
3	Lab time to understand system and code	31	39	31	0	52	39	9	0	55	27	18	0

Table 95: Time distribution among tasks by *Cluster*

		<25	25-50	50-75	>75
1	Lab time to understand system and model	35	40	18	8
2	Lab time to write state driver	0	28	53	19
3	Lab time to understand system and code	33	41	23	3

Table 96: Time distribution among tasks for all clusters, experiments and labs