The Interlocutory Tool Box: Techniques for Curtailing Coincidental Correctness

A thesis presented for the degree of
Doctor of Philosophy

Krishna Patel

Brunel University London

Department of Computer Science

2017

Abstract

Eliminating faults in software systems is important, because they can have catastrophic consequences. This can be achieved by testing and debugging. Testing involves executing the system with a test case to obtain an output. The output is evaluated against the tester's expectations; deviation from these expectations indicates that a fault has been detected. Debugging involves using information about the fault, that was gleaned during testing, to isolate the fault in the system. Coincidental correctness is a widespread phenomenon in which a fault corrupts a program state, and despite this, the system produces an output that satisfies the tester's expectations. Coincidental correctness can compromise the effectiveness of testing and debugging techniques.

This thesis investigated methods for alleviating coincidental correctness in testing and debugging. The investigation culminated in four techniques. The first technique is called Interlocutory Testing. Interlocutory Testing is a framework for the development of test oracles that are referred to as Interlocutory Relations. Interlocutory Relations are the first type of oracle that has been specifically designed to operate effectively in the presence of coincidental correctness.

Metamorphic Testing was pioneered for testing non-testable systems. However, the effectiveness of this technique can be compromised by coincidental correctness. The second technique, Interlocutory Metamorphic Testing, is a version of Metamorphic Testing that has been integrated with Interlocutory Testing, to alleviate the impact of coincidental correctness on Metamorphic Testing.

Interlocutory Mutation Testing is the third technique. This technique uses similar principles to Interlocutory Testing to alleviate the Equivalent Mutant Problem in the presence of coincidental correctness and non-determinism. Finally, the fourth technique is Interlocutory Spectrum-based Fault Localisation. This technique uses Interlocutory Relations to ameliorate the effects of coincidental correctness on fault localisation.

Each technique was empirically evaluated. The results were promising, and indicated that these techniques were capable of mitigating the impact of coincidental correctness.

Publications

- K. Patel and R. Hierons. Resolving the equivalent mutant problem in the presence of non-determinism and coincidental correctness. In *Proceedings of the 28th IFIP International Conference on Testing Software and Systems*, volume 9976 of *Lecture Notes in Computer Science*, pages 123 – 138, Graz, Austria, 17–19 October 2016. Springer International Publishing.[1]

[1] This paper is a preliminary version of Chapter 5.

Contents

1 Introduction **1**

1.1 Aim and Objectives 1

1.1.1 Aim 1

1.1.2 Objectives 1 and 2: Testing and Coincidental Correctness 1

1.1.3 Objective 3: The Equivalent Mutant Problem, Non-Determinism and Coincidental Correctness 4

1.1.4 Objective 4: Fault Localisation and Coincidental Correctness 5

1.2 Contributions 6

1.3 Outline 7

2 Background **9**

2.1 Review Protocol 9

2.1.1 Scope 9

2.1.2 Search 10

2.1.3 Data Extraction 12

2.1.4 Quality 12

2.1.5 Synthesis Overview 13

2.2 N-version Testing 15

2.2.1 Effectiveness 15

2.2.2 Usability 16

2.2.3 Cost 17

2.2.4 Content-based Image Retrieval 18

2.3 Metamorphic Testing 18

2.3.1 Effectiveness 19

2.3.2 Usability 22

2.3.3 Efficiency 24

2.3.4 Combination Relations 24

2.3.5 Metamorphic Runtime Checking 25

2.3.6 Semi-Proving 25

2.3.7 Heuristic Test Oracles 26

2.4 Assertions 26

2.4.1 Effectiveness 27

2.4.2 Usability 28

2.4.3 Multithreaded Programs 28

2.4.4 Further discussion . . . 29
2.5 Machine Learning . . . 29
2.5.1 Effectiveness . . . 29
2.5.2 Usability . . . 30
2.5.3 Metamorphic Machine Learning . . . 32
2.6 Statistical Hypothesis Testing . . . 32
2.6.1 Assumptions . . . 32
2.6.2 Effectiveness . . . 33
2.6.3 Statistical Metamorphic Testing . . . 34
2.7 Comparing techniques . . . 34
2.7.1 Effectiveness . . . 34
2.7.2 Efficiency . . . 36
2.7.3 Usability . . . 37
2.8 Coincidental Correctness . . . 37
2.8.1 Test case selection . . . 39
2.8.2 Mutation Testing . . . 40
2.8.3 Debugging . . . 41
2.9 Threats to validity . . . 44
2.9.1 Search . . . 44
2.9.2 Data Extraction . . . 44
2.9.3 Quality Criteria . . . 44
2.9.4 Throughout the process . . . 45
2.10 Conclusion . . . 45

3 Interlocutory Testing **47**
3.1 Interlocutory Testing — Technique Description . . . 47
3.1.1 Interlocutory Testing and Coincidental Correctness . . . 48
3.1.2 Interlocutory Testing and Non-determinism . . . 53
3.1.3 Multiple IRs . . . 56
3.2 Experimental Design . . . 56
3.2.1 Main Experiment . . . 57
3.2.2 Test Suite Experiment . . . 61
3.2.3 Feasibility Experiment . . . 63
3.3 Results and Discussion . . . 66
3.3.1 RQ1. Is Interlocutory Testing feasible? . . . 66
3.3.2 RQ2. How effective is Interlocutory Testing? . . . 67
3.3.3 RQ3. Do the net gains obtained from probabilistic IRs outweigh the potential net losses? . . . 69
3.3.4 RQ4. Which IRs should be prioritised? . . . 70
3.3.5 RQ5. How consistent is the effectiveness of Interlocutory Testing across different test suites? . . . 73
3.3.6 Discussion . . . 78
3.4 Related Work . . . 81

3.5 Threats to Validity . . . 83
3.5.1 Internal Validity . . . 83
3.5.2 External Validity . . . 86
3.5.3 Construct Validity . . . 87
3.5.4 Statistical Validity . . . 87
3.6 Conclusion . . . 88

4 Interlocutory Metamorphic Testing 91
4.1 Interlocutory Metamorphic Testing — Technique Description . . . 92
4.1.1 Intuition . . . 92
4.1.2 Interlocutory Metamorphic Testing . . . 93
4.2 Experimental Design . . . 94
4.2.1 Research Questions . . . 94
4.2.2 Subject Programs . . . 95
4.2.3 Test Cases . . . 95
4.2.4 Faults . . . 95
4.2.5 Interlocutory Metamorphic Relations . . . 96
4.3 Results and Discussion . . . 98
4.3.1 RQ1. Is Interlocutory Metamorphic Testing a feasible testing technique? . . . 98
4.3.2 RQ2. How effective is Interlocutory Metamorphic Testing in the presence of coincidental correctness? . . . 98
4.3.3 RQ3. What impact does Interlocutory Metamorphic Testing have on the effectiveness of Interlocutory Testing? . . . 101
4.3.4 RQ4. What effect does the test suite have on the effectiveness of IMT? . . . 104
4.3.5 Discussion . . . 105
4.4 Threats to validity . . . 106
4.5 Conclusions . . . 107

5 Interlocutory Mutation Testing 108
5.1 Interlocutory Mutation Testing — Technique Description . . . 109
5.1.1 Intuition . . . 109
5.1.2 Technique Description . . . 110
5.2 Experimental Design . . . 111
5.2.1 Research Questions . . . 111
5.2.2 Subject Programs . . . 111
5.2.3 Interlocutory Relations . . . 112
5.2.4 Mutants . . . 112
5.2.5 Test Cases . . . 113
5.2.6 Benchmark . . . 114
5.3 Results and Discussion . . . 114
5.3.1 RQ1. How Accurate is IMuT in the Presence of Coincidental Correctness and Non-determinism? . . . 114
5.3.2 RQ2: What Impact Might IMuT have on Productivity? . . . 117

5.3.3 RQ3: How does IMuT Compare to Other Mutant Classification Techniques? . 118
5.3.4 RQ4: Can Our Findings Revolving Around the Accuracy of IMuT Generalise to Deterministic Systems without Coincidental Correctness? 119
5.3.5 RQ5: Can Our Findings Revolving Around the Productivity Gains Offered by IMuT Generalise to Other Problem Domains? 120
5.4 Related Work 121
5.4.1 The Equivalent Mutant Problem and Non-Deterministic Systems 121
5.4.2 Weak Mutation Testing 121
5.5 Threats to Validity 122
5.6 Conclusion 123

6 Interlocutory Spectrum-based Fault Localisation 125
6.1 Interlocutory Spectrum-based Fault Localisation — Technique Description 126
6.1.1 Interlocutory Spectrum-based Fault Localisation: Deterministic IRs 126
6.1.2 Interlocutory Spectrum-based Fault Localisation: Probabilistic IRs 131
6.1.3 Applying Interlocutory Spectrum-based Fault Localisation 133
6.2 Experimental Design 133
6.2.1 Research Questions 133
6.2.2 Subject Programs 134
6.2.3 Test Cases 134
6.2.4 Faults 134
6.2.5 Interlocutory Relations 134
6.2.6 Measures 135
6.2.7 Benchmark Techniques 135
6.2.8 LOC and Logging 136
6.3 Results and Discussion 136
6.3.1 RQ1. Is ISBFL a feasible debugging technique? 136
6.3.2 RQ2. How effective is ISBFL at localising faults? 137
6.3.3 RQ3. What are the factors that affect the fault localisation effectiveness of ISBFL? 139
6.3.4 RQ4. How does the fault localisation effectiveness of ISBFL compare to other SBFL techniques? 143
6.3.5 Discussion 145
6.4 Threats to Validity 145
6.4.1 Internal Validity 145
6.4.2 External Validity 146
6.4.3 Construct Validity 146
6.5 Conclusions 147

7 Conclusions 149
7.1 Contributions 149
7.1.1 Mapping Study 150
7.1.2 Interlocutory Testing 150
7.1.3 Interlocutory Metamorphic Testing 151

7.1.4 Interlocutory Mutation Testing . . . 151
7.1.5 Interlocutory Spectrum-based Fault Localisation . . . 152
7.2 Limitations and Future Work . . . 152
7.2.1 Limitations of the Techniques . . . 152
7.2.2 Limitations of the Research . . . 153
7.2.3 Future work . . . 154
7.3 Summary . . . 154

8 References **157**

9 Appendices **175**
A Interlocutory Relations - Genetic Algorithm Subject Program . . . 175
B Real Faults . . . 188
C Interlocutory Relations - Dijkstra's Algorithm . . . 191
D Interlocutory Relations - Bubble Sort . . . 193
E Interlocutory Relations - Binary Search . . . 194
F Interlocutory Relations - Knuth-Morris-Pratt . . . 195

List of Figures

1.1 If Statement 6

3.1 Mutants and IRs Heat Map 68
3.2 Number of faults detected by deterministic and probabilistic IRs 70
3.3 Number of mutants detected by IRs (grouped by Components) 71
3.4 Number of unique mutants detected by IRs (grouped by Components) 71
3.5 Number of mutants killed by TS2 to TS30; results filtered by IR type. 74
3.6 Difference in performance of TS1's IRs, against the corresponding IRs in TS2 to TS30. 75
3.7 Cumulative Frequency Graph 76
3.8 Difference between TS1's FPRs and the FPRs of TS2 to TS30. 77
3.9 Cumulative Frequency Graph. 77
3.10 Difference between TS1's FDRs and the FDRs of TS2 to TS30. 78

4.1 Number of mutants killed by each Probabilistic IR in both techniques 102
4.2 IRs that are members of one IMR and their mutation scores 103
4.3 IRs that are members of multiple IMRs and their mutation scores 103

5.1 Number of mutants that were correctly classified by Deterministic IRs, broken down by mutant type 115
5.2 Number of mutants that were correctly classified by Probabilistic IRs, broken down by mutant type 115
5.3 Summary of results 119

6.1 Sample Program Fragment 129
6.2 ISBFL's EXAMBest, EXAMAverage, and EXAMWorst measures for Bubble Sort, Dijkstra's Algorithm and Knuth-Morris-Pratt 137
6.3 ISBFL's EXAMBest, EXAMAverage, and EXAMWorst measures 138
6.4 Box Plots of the EXAMAverage of standard and coincidentally correct faults 139
6.5 Minimum, Average, and Maximum EXAMAverage of each IR(s) 142
6.6 Minimum, Average and Maximum EXAMAverage by class 142
6.7 Difference in EXAMAverage between ISBFL and the benchmark techniques 143
6.8 Number of test cases that are classified as failed by DISBFL and SBFL. 144

List of Tables

2.1 Relevance Inclusion and Exclusion Criteria 11
2.2 Search Strings 12
2.3 Data Extraction Form 13
2.4 Quality Instrument 14
2.5 A summary of the effectiveness data for each technique, based on Sections 2.2 to 2.6. . 35
2.6 A summary of the efficiency data for each technique, based on Sections 2.2 to 2.6. . . 37
2.7 A summary of the usability data for each technique, based on Sections 2.2 to 2.6. . . . 38

3.1 Descriptive statistics of the mutation scores obtained by TS2 to TS30, and a summary of the TS1's mutation scores. Standard Faults, Coincidentally Correct Faults, and Both Faults are represented by SF, CCF, and BF respectively. 75

4.1 IMT's Mutation Scores 99
4.2 Pairwise comparisons between each IMR, based on Fisher's Exact Test 99
4.3 Number of unique faults found by IMR, broken down by fault type 100
4.4 Overview of Interlocutory Testing and IMT's results 101

6.1 Descriptive statistics of each cluster, based on EXAMAverage 138

List of Abbreviations

(BVA) Boundary Value Analysis

(DET) Deterministic Execution Testing

(DISBFL) Deterministic Interlocutory Spectrum-based Fault Localisation

(FDR) Failure Detection Rate

(ID) Interlocutory Decision

(IMR) Interlocutory Metamorphic Relation

(IMT) Interlocutory Metamorphic Testing

(IMuT) Interlocutory Mutation Testing

(IOR) Input-Output Relationship

(IR) Interlocutory Relation

(ISBFL) Interlocutory Spectrum-based Fault Localisation

(IT) Interlocutory Testing

(LOC) Line of Code

(MET) Multiple Execution Testing

(ML) Machine Learning

(MO) Mutation Operator

(MR) Metamorphic Relation

(MS) Mutation Score

(MT) Metamorphic Testing

(MTG) Metamorphic Test Group

(PIR) Probabilistic IR

(PISBFL) Probabilistic Interlocutory Spectrum-based Fault Localisation

(RI) Reference Implementation

(SBFL) Spectrum-based Fault Localisation

(SHT) Statistical Hypothesis Testing

(SLOC) Source Lines of Code

(SUT) System Under Test

(TEMDT) Traditional Equivalent Mutant Detection Technique

(TSO) Tournament Selection Operator

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor Robert M. Hierons, for his unwavering support throughout my studies. His frequent, detailed feedback on drafts, guidance and advice on the academic process, and the insightful discussions we shared were instrumental in the production of this thesis and my personal development. It was a privilege to work with such a helpful, dedicated, and understanding person.

I would also like to thank my second supervisor, Professor Martin Shepperd, Professor Tracy Hall, Dr Steve Counsell, and Dr Andrea Capiluppi for their suggestions on several aspects of the thesis. I'm grateful to all members of the Department of Computer Science at Brunel University for being such a welcoming, warm, and encouraging community. It was a pleasure to work in such a positive environment.

I also appreciate the assistance of the global research community. In particular, I would like to thank Professor Jeff Offutt, and Lin Deng for helping me with the MuJava tool, the authors of the papers that were processed by our Mapping Study, who assisted us through email correspondence, and the anonymous reviewers for their helpful comments on our chapter submissions.

Finally, I would like thank my parents, Vinod Patel and Parvati Patel, and my brother, Bhavesh Patel, for their endless love, encouragement, and support.

Chapter 1

Introduction

This chapter outlines the problems being tackled by the thesis, as well as our aim and objectives (see Section 1.1). It also presents the high level contributions and outline of the thesis. These can be found in Sections 1.2 and 1.3 respectively.

1.1 Aim and Objectives

The aim of this thesis is to alleviate the impact of coincidental correctness in testing and debugging. To address this aim, the thesis will attempt to fulfil the following objectives:

- Develop a new testing technique that can operate effectively in the presence of coincidental correctness.
- Modify Metamorphic Testing to reduce its susceptibility to coincidental correctness.
- Develop a partial solution to the Equivalent Mutant Problem that can tolerate coincidental correctness and non-determinism.
- Modify Spectrum-based Fault Localisation, to mitigate the impact of coincidental correctness.

The remainder of this section fleshes out the aim and objectives further.

1.1.1 Aim

Coincidental correctness describes a situation in which a corrupt program state manifests in the system under test (SUT), but despite this the SUT arrives at an output that could have been produced by a correct version of the SUT[1]. Coincidental correctness is widespread [119], and can reduce the effectiveness of various testing and debugging activities and techniques. This thesis aims to counteract this reduction in effectiveness.

1.1.2 Objectives 1 and 2: Testing and Coincidental Correctness

A developer might make a mistake when writing source code that could ultimately cause the system to behave incorrectly. In such a scenario, the developer's action is called an error, the mistake in the source code is referred to as a fault, bug or defect, and the system's incorrect behaviours are described

[1] We refer to such an output as a "plausible output".

as a failure. Failures can have catastrophic consequences [79]; it's therefore important to verify the correctness of software. This can be achieved by testing. A test case is a set of values that collectively form an input for the system under test (SUT), and a test suite is a set of test cases. Testing involves generating a test suite for the SUT, and for each test case in the test suite, predicting the SUT's outcome for this test case, executing the SUT with this test case to obtain an output, and finally comparing this output to the predicted outcome. Comparisons that reveal discrepancies between the output and predicted outcome are interpreted as evidence that the SUT is faulty.

The testing process described above has three particularly important components; test case generation, the testing methodology, and test oracles. The coverage of a test case refers to the proportion of the source code that is executed by that test case, and the coverage of a test suite is the proportion of the source code that is executed by the test cases within that test suite. Different test cases can either execute the same or different source code, which means that one's choice of test cases has an impact on the overall coverage of the test suite, and therefore the likelihood of executing a fault. Test case generation is responsible for selecting test cases to be included in the test suite.

The second important component of the testing process is the precise testing methodology that is used. In particular, whether functional testing/black box-testing or structural testing/white box testing is applied. A white box testing approach assesses the correctness of the SUT based on the SUT's internal program states, whilst black box testing approaches solely base their assessments on input-output pairs.

The third important component is test oracles. Test oracles are an integral part of testing; they are responsible for the outcome prediction and comparison tasks outlined above. Unfortunately, testing has a drawback related to test oracles, called the oracle problem. The oracle problem[2] describes a testing scenario, in which it is infeasible to predict the test outcome or perform the comparison task discussed above [190]. A sizeable amount of research has been conducted on alleviating the oracle problem. We conducted a Mapping Study on the oracle problem, which can be found in Chapter 2. The Mapping Study revealed a suite of techniques that can operate effectively in non-testable systems. Assertions is an example of such a technique [74].

The Mapping Study also highlighted a related problem — coincidental correctness. Coincidental correctness describes a phenomenon in testing, where a fault affects the behaviour of the SUT, but the final output produced by the SUT is plausible [119]. To illustrate, let $Prog$ be a program that permutes the order of an array of positive integers ($List$). $Prog$ iterates multiple times. On each iteration, variables I and J are assigned a random array index from $List$ ($I = J$ is possible). The values stored at $List.get(I)$ and $List.get(J)$ are swapped. Let $Prog_f$ be a faulty version of $Prog$ that does not swap these values, if either I or J refers to the last index of $List$. The output produced by $Prog_f$ is plausible, since in $Prog$, $List.get(I) = List.get(J)$ is possible, which means that a swap may also not occur in $Prog$.

As discussed above, testing techniques report failures if a discrepancy between the actual output and predicted outcome is revealed. Since the actual output can be plausible when coincidental correctness is present, regardless of whether faulty behaviour manifests, it can match the predicted outcome, and thus the testing process can erroneously conclude that the system is correct. To illustrate, an Assertion may check that every integer in $List$ is a positive number; such a check cannot

[2]The research community interchangeably refers to systems that have the oracle problem as non-testable systems.

detect $Prog_f$. Coincidental correctness can clearly compromise the effectiveness of traditional testing techniques. It's therefore not surprising that the Mapping Study found that most of the techniques that were covered were not particularly effective in the presence of coincidental correctness, and that there was no empirical evidence to demonstrate that the remaining techniques were effective under these circumstances.

Various studies [119, 120, 3, 197] have concluded that coincidental correctness is widespread (see Section 2.8 for a more detailed discussion). Despite the prevalence and importance of coincidental correctness, relatively little research has been conducted on alleviating coincidental correctness in testing. To the best of our knowledge, all of the solutions that have been proposed largely focus on test case selection; there are no solutions that incorporate test oracles (see Section 2.10). This motivates our first objective; to develop such a solution.

We observed that the relationship between the input and output of the SUT, in conjunction with one's knowledge/expectations about the SUT, can be used to predict aspects of the execution trace. To illustrate, consider the permutation program example above. Let *Input* and *Output* be the state of *List* before and after *Prog* has processed it respectively. If $Input = Output$, we can predict the following aspects of the execution trace. Let $List_{iter}$, I_{iter}, and J_{iter} respectively denote the state of *List*, *I* and *J* at the start of iteration *iter*. For array index *K* in *Output*, let *RelevantIter* be the last iteration in which $I_{RelevantIter} = K$ or $J_{RelevantIter} = K$. We can predict that if $I_{RelevantIter} = K$, then $List.get(J_{RelevantIter}) = Input.get(K)$. Otherwise, if $J_{RelevantIter} = K$, then $List.get(I_{RelevantIter}) = Input.get(K)$. The correctness of this prediction is predicated on whether the SUT's behaviour mirrors the tester's expectations. To illustrate, suppose that the permutation program has the aforementioned fault $Prog_f$. $Prog_f$ can lead to situations in which $Input = Output$, $I_{RelevantIter} = K$, and $List.get(J_{RelevantIter}) \neq Input.get(K)$. It can also create situations in which $Input = Output$, $J_{RelevantIter} = K$, and $List.get(I_{RelevantIter}) \neq Input.get(K)$. The failure to satisfy the prediction above in such situations shows that the behaviour of the SUT does not satisfy the tester's expectations i.e. a fault is present. This demonstrates that an approach that leverages the relationship between the input and output to make predictions about aspects of the execution trace can be used for testing in the presence of coincidental correctness. In Chapter 3, we introduce a technique called Interlocutory Testing, which is an implementation of such an approach. Thus, this attempts to address our first objective.

Another interesting observation made by the Mapping Study is that Metamorphic Testing (MT) was one of the most studied techniques, in the context of the oracle problem. In MT, two sets of test cases are executed. Those in the first set are referred to as source test cases. These test cases can be generated by any test case selection strategy [35]. Those in the second set are called follow up test cases; these test cases are generated based on specific source test cases and a metamorphic property [135]. A metamorphic property is an expected relationship between the outputs of source and follow up test cases. A Metamorphic Relation (MR) is an oracle that is based on MT, and is evaluated by checking that this relationship holds. For example, let $Max(Ints)$ be a program that returns the maximum value of a list of integers, *Ints*. An MR may generate one Source Test Case ($SourceTestCase$) and one Follow-up Test Case ($FollowupTestCase$), such that $FollowupTestCase$ contains two of each of the integers in $SourceTestCase$ e.g. $SourceTestCase = \langle 3, 4, 5 \rangle$ and $FollowupTestCase = \langle 3, 3, 4, 4, 5, 5 \rangle$. The MR may check that the output of $Max(SourceTestCase) = Max(FollowupTestCase)$. As men-

tioned above, none of the testing techniques that were covered by the Mapping Study were empirically shown to be effective in the presence of coincidental correctness, and this includes MT [20, 202]. This inspired our second objective: to alleviate the impact of coincidental correctness on MT. Chapter 4 extends MT with Interlocutory Testing to create a new testing technique called Interlocutory Metamorphic Testing. The primary goal of Interlocutory Metamorphic Testing is to enable MT to operate effectively in the presence of coincidental correctness, and thus to address the second objective.

1.1.3 Objective 3: The Equivalent Mutant Problem, Non-Determinism and Coincidental Correctness

Information about the effectiveness of a testing technique can inform its design, development, and application. The process of obtaining this information would ideally involve exercising the testing technique on a set of real faults and recording the proportion of faults that were detected. Unfortunately, this is usually infeasible because of a lack of fault data. Mutation Testing alleviates this by generating artificial faults [142], which are intended to be reasonably accurate simulations of real faults [2].

Mutation Testing operates by applying a minor augmentation (referred to as a mutation) to the SUT (S_o) to produce a new version (S_m) [21] called a mutant. For example, a statement $X < 5$ in S_o might be transformed into $X > 5$ in S_m. Mutations are intended to modify the behaviour of the SUT to simulate a fault. A testing technique can then be applied to a mutant to deduce whether it can detect the fault or not; if it can, then the mutant is said to have been killed, otherwise the mutant is considered to have survived. Let KM be the total number of killed mutants and TM denote the total number of mutants. The effectiveness of a testing technique can be measured with the mutation score formula: $\frac{KM}{TM}$.

Unfortunately, Mutation Testing has several limitations; its propensity to produce equivalent mutants being the most prominent — this is known as the Equivalent Mutant Problem [86]. A mutant, S_m, is said to be an equivalent mutant, if S_m is semantically equivalent to the SUT, S_o. For example, suppose that $Math.abs(5)$ and $Math.abs(-5)$ appear on Line 1 in S_o and S_m respectively. S_m is an equivalent mutant because the change doesn't modify the behaviour of S_o. Other examples include mutant injection into dead code or mutants that cause performance improvements [86].

Equivalent mutants are problematic because they can skew empirical results, if one does not identify and exclude them from experiments. In a study conducted by Yao et al. [199], it was discovered that equivalent mutants are produced for most programs, regardless of size. Despite the fact that deducing mutant equivalence is undecidable [19], the prevalence of the equivalent mutant problem has motivated the development of several techniques that attempt to automate the classification of mutants as equivalent or non-equivalent. Let $S_o(I)$ and $S_m(I)$ denote the respective outputs of S_o and S_m for a given input I. One commonly used method involves exposing S_o and S_m to a test suite to obtain a set of pairs $\langle S_o(I), S_m(I) \rangle$ and assumes that S_o and S_m are equivalent if the following condition holds for each pair: $S_o(I) = S_m(I)$. For ease of reference, we refer to this as the Traditional Equivalent Mutant Detection Technique (TEMDT). An example of the use of TEMDT can be found in a study conducted by Sadi et al. [165].

However, this assumption doesn't always hold. For example, the presence of coincidental correctness can lead to situations in which S_o and S_m produce the same output, even if S_m is semantically

different from S_o. This can lead to non-equivalent mutants being mistakenly classified as equivalent and bias the results because it could lead to the eradication of certain fault types from the evaluation. If the technique is particularly adequate for or struggles with these fault types, then the effectiveness measurements could either be an underestimate or overestimate respectively. The assumption may also not withstand non-determinism; non-deterministic behaviours may be responsible for discrepancies in the outputs of S_o and S_m, but one may misconstrue the source of these discrepancies as having originated from the mutation [21]. Alternative techniques have been proposed to address these problems, but have limitations (see Sections 2.8.2 and 5.4). Manual inspection is typically necessary under such circumstances [1], and can be very costly. For example, Zeller and Schuler [203] manually classified 140 mutants and reported that on average, 14 minutes and 28 seconds was required per mutant classification. Thus, our third objective is to devise an automated means of alleviating the Equivalent Mutant Problem in situations in which coincidental correctness and/or non-determinism is present.

The $Prog_f$ example in Section 1.1.2 demonstrates that the intuition behind Interlocutory Testing can be used to distinguish between coincidentally correct and correct behaviours. Since the example also implements non-determinism, it also shows that the intuition can tolerate non-deterministic behaviours. Thus, it may be possible to leverage this intuition to ameliorate the Equivalent Mutant Problem, for non-deterministic systems that are susceptible to coincidental correctness, as follows. Let S denote the SUT and M be a mutated version of S. Suppose that M is executed with an input ($MInput$), and produces an execution trace (MET), and an output ($MOutput$). The relationship between $MInput$ and $MOutput$ can be used in conjunction with the tester's knowledge about S to predict aspects of MET. If this prediction is incorrect then this suggests that M is not an equivalent mutant. We call this approach Interlocutory Mutation Testing. Chapter 5 investigates whether Interlocutory Mutation Testing could be an effective solution for the Equivalent Mutant Problem, in the presence of coincidental correctness and non-determinism. Interlocutory Mutation Testing attempts to address Objective 3.

1.1.4 Objective 4: Fault Localisation and Coincidental Correctness

Fault Localisation is a debugging task that involves using one's knowledge about the SUT and a failure to deduce which line of code (LOC) in the SUT is faulty. This is one of the most expensive tasks in debugging [107] and has therefore inspired a lot of research into how this task could be automated. One of the most promising techniques that has arisen from this collective effort is Spectrum-based Fault Localisation (SBFL) [195].

SBFL instruments the SUT with predicates that evaluate to true when noteworthy events occur in an execution trace. The SUT is then executed with a test suite to produce a set of execution traces (ET) based on these predicates. ET is partitioned into two subsets ET_p and ET_f, such that ET_p and ET_f contain the execution traces of passed and failed test cases respectively. The frequency with which each predicate evaluates to true in ET_f and ET_p is used as evidence that the event that is represented by the predicate under consideration is correlated with failure, or the converse respectively. Various mathematical formula are available e.g. Ochiai, Tarantula and Jaccard [52]; one of these can be used to leverage the aforementioned frequency information to determine a "suspiciousness score" for each predicate i.e. the likelihood of each predicate being correlated with failure.

```
10.   x = 5
11.   if (x >y){
12.       x = x + 1
13.       y = y + 2
14.   } else {
15.       x = x + 2
16.       y = y + 1
17.   }
```

Figure 1.1: If Statement

Each predicate maps to a set of LOC, such that the event that is captured by the predicate is dependent on the behaviours of this set of LOC. The suspiciousness score of each predicate is appended to their set of LOC. For example, consider Figure 1.1. A predicate p_1 may be based on the outcome of the if statement being true and thus map on to Lines 10 to 13 (inclusive) but not 14 to 17 (inclusive). Thus if p_1 obtained a suspiciousness score of 0.7, this will be reflected on Lines 10 to 13 (inclusive). The LOC in the SUT are finally sorted in descending order based on suspiciousness and this is reported to the tester.

Ideally, a faulty LOC will be towards the top of this list i.e. have a higher than average suspiciousness score. This enables testers to prioritise their investigation of each LOC, based on the likelihood of it being faulty. Although SBFL has had some promising results, it has also been found to perform poorly when coincidental correctness is present [191] because test cases that execute faults may be misinterpreted as passing testing cases [119] and this can confound the suspiciousness score calculations. Our fourth objective is to mitigate this problem.

Chapter 6 introduces a new variant of SBFL, called Interlocutory Spectrum-based Fault Localisation (ISBFL), in which the predicates used by SBFL are supplemented with oracles based on Interlocutory Testing. The primary goal of ISBFL is to enable effective fault localisation in the presence of coincidental correctness. ISBFL aims to address Objective 4.

1.2 Contributions

The main, high level contributions of the thesis are:

1. A comprehensive Mapping Study on the oracle problem.
2. A new testing technique called Interlocutory Testing that can operate effectively in the presence of coincidental correctness, non-determinism, and the oracle problem.
3. An extended version of Metamorphic Testing that is less susceptible to the effects of coincidental correctness. This extended version is referred to as Interlocutory Metamorphic Testing.
4. A partial solution for the Equivalent Mutant Problem, called Interlocutory Mutation Testing, that can perform accurate classifications despite the presence of coincidental correctness and/or non-determinism.
5. Interlocutory Spectrum-based Fault Localisation; a fault localisation technique that can provide debugging support when coincidental correctness is present.

1.3 Outline

1. Chapter 2 presents our Mapping Study on the oracle problem. It also presents background material on coincidental correctness. Background material that is only relevant to specific chapters is delegated to those chapters.

2. Chapter 3 introduces Interlocutory Testing. It also presents the results of a series of experiments that attempted to ascertain the effectiveness and generalisability of Interlocutory Testing. A comparative analysis between Interlocutory Testing and several other testing techniques in terms of their effectiveness and usability is also given in this chapter.

3. Chapter 4 introduces Interlocutory Metamorphic Testing, and exercises it in several experiments to deduce its effectiveness and generalisability.

4. Chapter 5 introduces Interlocutory Mutation Testing, and investigates its effectiveness, generalisability, and the productivity gains that can be ascertained by leveraging the technique. An experiment that compares Interlocutory Mutation Testing to TEMDT is also presented.

5. Interlocutory Spectrum-based Fault Localisation is introduced in Chapter 6, along with experiments that illustrate its effectiveness.

6. Finally, Chapter 7 outlines our conclusions, discusses the limitations of our work, and presents possible future research avenues.

Chapter 2

Background

As mentioned in Section 1.1.2, we conducted a Mapping Study on the Oracle Problem. The Mapping Study surveyed literature on automated testing techniques that can detect functional software faults in non-testable systems, and endeavoured to leverage this information to satisfy the following objectives. The first objective was to present a series of discussions about each of these techniques, from different perspectives e.g. effectiveness, usability, and efficiency. The second objective was to perform a series of comparisons between these techniques, based on effectiveness, efficiency and usability. The final objective was to identify research opportunities. Further details about the Mapping Study can be found in the Review Protocol, in Section 2.1. This chapter presents background material on the Oracle Problem, in the form of the outputs of this Mapping Study. In particular, Sections 2.2 to 2.6 present the aforementioned discussions about each technique, and Section 2.7 conducts a series of comparisons between the techniques. Potential research opportunities are identified throughout these sections.

This chapter also documents background material on Coincidental Correctness, which can be found in Section 2.8. Finally, we close this chapter by presenting threats to validity in Section 2.9, and drawing conclusions and summarising our contributions in Section 2.10.

2.1 Review Protocol

To conduct our Mapping Study, we first defined a Review Protocol, based on the guidelines of Kitchenham [99], Popay et al. [156], Higgins et al. [76], and Shepperd [172]. This section presents our Review Protocol. In particular, Sections 2.1.1, 2.1.2, 2.1.3 and 2.1.4 outline the scope, search procedure, data extraction approach, and quality appraisal methodology respectively. Finally, a brief overview of the synthesis, which forms the majority of the remainder of this chapter, is presented in Section 2.1.5.

2.1.1 Scope

The scope of this Mapping Study was originally *automated testing and debugging* techniques that have been designed to detect *functional software* faults in *non-testable systems.* Our Review Protocol (presented in Sections 2.1.2 to 2.1.4) is based on this scope. We decided to narrow the scope of our synthesis (i.e. Sections 2.1.5 to 2.7) to enhance the focus of the Mapping Study. In particular, our revised scope does not consider debugging techniques. We realised that Specification-based Testing

and Model-based Testing depend on the availability of a specification or model, and that the oracle problem implies that these are not available. To that end, we also decided to omit these techniques from the scope of our synthesis.

2.1.2 Search

A paper is considered to be relevant if it adheres to the Inclusion and Exclusion Criteria listed in Table 2.1. To check a paper against these criteria, we adopted an iterative process, where successive iterations checked the paper in escalating levels of detail [172] against the Inclusion and Exclusion Criteria; if enough evidence could be accrued during an early iteration to classify the paper, then the process terminated prematurely. The iterations were as follows: {title}, {abstract, introduction, conclusion}, {the entire paper}. We conducted a search in mid 2014 to find relevant papers (that were available before and during mid 2014). We achieved this by applying several search methods in parallel and iteratively, and checking the relevance of each search result returned by these search methods, by using the aforementioned iterative process. The remainder of this section outlines these search methods.

One of our methods included using the search strings listed in Table 2.2 to query six research repositories: Brunel University Summon, ScienceDirect (using the Computer Science Discipline filter), ACM (queried using Google's "site:" function), IEEE, Google (twice — with the filter on, and off), and Citeseerx (each search term prefixed with "text:"). Let $Results_{SS}^{RR}$ denote the papers that were returned by a research repository, RR, in response to a search string, SS. It would have been infeasible to manually check the relevance of all of the papers in $Results_{SS}^{RR}$. Thus, we used the following terminating condition: the first occurrence of 50 consecutive irrelevant results. We retained papers that were found to be relevant before the terminating condition was satisfied.

During the search process, we became aware of several techniques that had been used to solve the oracle problem. The authors postulated that other studies on these techniques in the context of the oracle problem may also have been conducted. Thus, a specialised search string for each technique was prepared; these search strings supplemented those in Table 2.2.

Every paper in the reference list of each relevant paper was also checked for relevance. Again, we retained papers that were found to be relevant.

We compiled a list of all of the authors that had contributed to the papers that were deemed to be relevant. Each of these authors had at least one list of their publications publicly available. Examples of such lists include: author's personal web page, CV, DBLP, Google Scholar, and the repository the author's study was originally discovered in. We selected one list per author, based on availability and completeness, and checked all of the publications on this list for relevance.

Finally, all of the authors were emailed a list of their papers that had been discovered by the search, accompanied with a request for confirmation regarding the completeness of the list. This enabled the procurement of cutting edge works in progress, and also reduced publication bias [99].

The various search methods described above led to the discovery of several papers that we did not have access to. We were able to obtain some of these papers by contacting the authors. The rest of these papers were omitted from the Mapping Study. The search methods also returned what we believed were duplicate research papers. The authors of these papers were asked to confirm our suspicions, and duplicates were removed.

Issue	**Criteria**
Problem Domain	*The targeted problem domain/context in which testing is undertaken must be non-testable.*
	The problem itself must revolve around the lack of a mechanism to judge the correctness of an output.
	The existence of the non-testable aspect must not be considered to be a fault in itself.
	The non-testable characteristic experienced in the SUT must arise from the software.
	The types of faults considered by the paper must be software faults.
	The quality attribute of the system being tested must be functional correctness.
Solution Space	*The paper must include some sort of solution to the problem e.g. a testing technique.*
	The primary solution to the problem must revolve around an automated fault detection mechanism.
	The solutions must fall under the domains of either testing or debugging.
Paper Type	*Journal articles, conference proceedings, technical reports, book chapters, and magazines must be included.*
	Papers that have a broad focus e.g. frameworks or systematic reviews must contribute a relatively substantial amount of relevant content. For example, a paper is not deemed to be relevant if all of its relevant material is comprised of a short aside.
	Duplicates must be excluded. We consider rewrites and preliminary/older versions of the same papers to be duplicates. We also consider journal papers that extend conference papers to be duplicates, as well as published chapters of theses. Preference is given to published over non-published papers, the most up-to-date version, and the paper from the most reputable source. If both papers are published in reputable journals, the most detailed one is taken, and in the case that they are precise duplicates of each other, an arbitrary choice is made.
	The paper must be written in English.
	The paper must be accessible.
	The paper must have been published before mid 2014.

Table 2.1: Relevance Inclusion and Exclusion Criteria

In total, our search methods collectively procured 141 papers.

Search Strings
((Stochastic OR (Non-deterministic OR nondeterministic OR “non deterministic” OR non-determinism OR nondeterminism OR “non determinism”)) AND (System OR Software OR Program OR Application OR Algorithm) AND Testing
((Stochastic OR (Non-deterministic OR nondeterministic OR “non deterministic” OR non-determinism OR nondeterminism OR “non determinism”)) AND (System OR Software OR Program OR Application OR Algorithm) AND ((“Check” OR “Checking”) OR (“Verification” OR “Verify”))
((Stochastic OR (Non-deterministic OR nondeterministic OR “non deterministic” OR non-determinism OR nondeterminism OR “non determinism”)) AND (System OR Software OR Program OR Application OR Algorithm) AND (“Fault Localisation” OR “Fault Localization”)
(“Random output” OR “Randomised output” OR “Randomized output” OR “Randomized algorithm” OR “Randomised algorithm”) AND (Systems OR Software OR Programs OR Applications OR Algorithms) AND Testing
(“Probabilistic System” OR “Probabilistic Program” OR “Probabilistic algorithm”) AND ((“Check” OR “Checking”) OR (“Verification” OR “Verify”))
((“NonTestable” OR “Non-Testable” OR “Non Testable”) OR (“Oracle Problem” OR “no oracle”) OR (“Pseudo-oracle” OR “Pseudo oracle”)) AND (Testing OR ((“Check” OR “Checking”) OR (“Verification” OR “Verify”)) OR (“fault localisation” OR “Fault localization”) OR (“Debugging” OR “Debug”) OR (“Fault detection” OR “Failure detection” OR “Mutant detection” OR “Defect detection” OR “Detecting Faults” OR “Detecting Failures” OR “Detecting Mutants” OR “Detecting Defects”) OR (“Validating” OR “Validate”))

Table 2.2: Search Strings

2.1.3 Data Extraction

We used the data extraction form in Table 2.3 to capture data from relevant papers, that was necessary to appraise study quality and address the research aims. Unfortunately, many papers did not contain all of the required data; thus requests were sent to authors to obtain missing data. Where this approach was unsuccessful, assumptions were made based on the paper and the author’s other work. For example, if they had not reported the number of mutants used, but tended to use 1000+ in other papers, one can assume a significant number of mutants were used in the study.

2.1.4 Quality

Our quality instrument is presented in Table 2.4. Each relevant paper was checked against this quality instrument. Papers that were found to have severe methodical flaws, and to have taken minimal steps to mitigate bias were deemed to be of low quality. Relatively little research has been conducted on the oracle problem; thus, many relevant studies are exploratory. Certain study design choices may have been unavoidable in such studies, and may cause a quality instrument to label these studies as low quality. This means that these valuable studies could be rejected, despite the fact that they may have been at the highest attainable quality at the time. To account for this, papers that were deemed to be of low quality, were only discarded if they did not make a novel contribution. This led to the

Data Extraction Form	
ID:	**Paper Title:**
Question	**Answer**
What evidence is there of there being a sufficient amount of relevant information in the paper to make a meaningful contribution to the Mapping Study's findings?	
What evidence is there to suggest the parameters of the experiment were representative and were they adequate described?	
What evidence is there to suggest the experimental set-up, conduct and experiment output analysis was appropriate, robust and unbiased?	
Have adverse effects been reported, and if so, how were they mitigated?	
Are the arguments compelling, critical and supported by internal and external evidence?	
Executive Summary:	
Noteworthy points made about Technique 1:	
Noteworthy points made about Technique 2:	
Noteworthy points made about Technique n:	

Table 2.3: Data Extraction Form

elimination of 4 papers. Thus a total of 137 papers were deemed to be suitable for our synthesis.

2.1.5 Synthesis Overview

Synthesis involves analysing and explaining the data that was obtained by the data extraction form to address the research aims. Narrative Synthesis was used because it is ideal for theory building [172] and explanations. The synthesis was conducted according to the guidelines of Popay et al. [156], Cruzes and Dyba [50], Silva et al. [51] and Barnett-Page and Thomas [13]. See Sections 2.2 to 2.7.

The Mapping Study process revealed that five umbrella testing techniques have been developed to alleviate the oracle problem — N-version Testing, Metamorphic Testing, Assertions, Machine Learning, and Statistical Hypothesis Testing. Thus, our synthesis focuses on these techniques. The research community has conducted a different amount of research on each technique, in the context of the oracle problem. For example, Metamorphic Testing has received more attention than any other testing technique. Naturally, the amount of attention that is afforded to each technique, by our synthesis, was determined by the amount of research that was conducted on that technique.

The disproportionate attention that has been given to Metamorphic Testing suggests that this technique may have numerous interesting research avenues. Although less attention has been afforded to the other techniques, they are known to be effective for some situations in which Metamorphic

Questions	**Answers**
Q1. Does the SUT have the non-testable characteristics that are being studied? *This question is not applicable to "Demonstration Papers" or "Extreme Depth of Analysis Papers" See below for their definitions.*	
Q2. Can you identify sources of potential bias in the paper, and are there any elements of the study design that can minimise bias or justify leaving the source of bias unchecked? *This question is not applicable to "Demonstration Papers" or "Extreme Depth of Analysis Papers" See below for their definitions.*	
Q3. If the author used measures or made inferences that are sensitive to the number of mutants, did they use an appropriate number of mutants? *This question is not applicable to "Demonstration Papers" or "Extreme Depth of Analysis Papers" See below for their definitions.*	
Q4. If the author used measures or made inferences that are sensitive to the number of test cases, did they use an appropriate number of test cases? *This question is not applicable to "Demonstration Papers" or "Extreme Depth of Analysis Papers" See below for their definitions.*	
Q5. If the author used measures or made inferences that are sensitive to the number of participants, did they use an appropriate number of participants? *This question is not applicable to "Demonstration Papers" or "Extreme Depth of Analysis Papers" See below for their definitions.*	
Q6. Were the authors' arguments and inferences backed up by internal and external evidence?	
Q7. Did the authors' use of language suggest they were biased towards a specific technique/findings e.g. were positive comments given about the comparison intervention or negative comments about the authors own technique?	
Q8. Was the authors' study novel? E.g. Extensions, using participants, including non-effectiveness measures etc.	
Comments:	
Definitions	
A demonstration paper is one that conducts an experiment, but not for the purpose of assessing the quality of the technique, but rather, to illustrate how it works/that it does work. One of the key features of a demonstration paper is that the experiment is not set-up to be rigorous, but to instead be an effective communication tool - i.e. simple and intuitive.	
"Extreme depth of analysis" papers are papers where either the results themselves have fairly complex associations amongst each other e.g. every pairwise combination of each test case and mutant has been individually presented, or if the author has a fairly extensive discussion about all of the individual cases.	

Table 2.4: Quality Instrument

Testing is not (see Section 2.7). Thus, the number of pages does not reflect how promising they are. However, it does mean that it is unlikely that all of the useful research avenues that are associated with these techniques have been explored.

Sections 2.2 - 2.6 present a series of discussions about each technique, and Section 2.7 compares these techniques. The discussions pertaining to each technique are organised into a set of high level issues e.g. effectiveness, efficiency, and usability. Some terms that are used to describe certain issues by one research community may be used to describe different issues by other research communities. We would therefore like to clarify how such terms are used in this chapter; in particular efficiency and cost. Efficiency is used to describe the amount of computational resources that are consumed or the amount of time required to perform a task, whilst cost is used in reference to monetary costs. Although effort/manual labour can be discussed in the context of cost, effort/manual labour is presented as a usability issue in this chapter.

2.2 N-version Testing

Let S be the SUT. Another system, S_R, is said to be a reference implementation (RI) of S, if it implements some of the same functionality as S. In N-version Testing, S and S_R are executed with the same test case, such that this test case executes the common functionality in these systems. The outputs of S and S_R that result from these executions are compared. N-version Testing reports a failure, if these outputs differ [190]. If one has access to multiple RIs, then this process can be repeated once for each RI.

N-version Testing was originally developed to alleviate the oracle problem. One form of oracle problem includes situations where the test outcome is unpredictable. Such an oracle problem can arise if the SUT has been designed to discover new knowledge [190] e.g. machine learning algorithms. Since an RI mimics the SUT to generate its own output, N-Version Testing does not require the tester to have prior knowledge about the test outcome. This makes it viable for such oracle problems.

2.2.1 Effectiveness

N-version Testing is fundamentally a black-box testing technique. It's therefore not surprising that some have found that N-version Testing cannot test the flow of events [139], and cannot detect certain fault types e.g. coincidental correctness [18], since white-box oracle information is necessary to achieve these feats. Let S be a system. In the future, S may be modified due to software maintenance. S' denotes the modified version of S. Faults may be introduced into S' during maintenance. Spectrum-based Fault Localisation is a debugging technique that represents the system's execution trace as program spectra. Tiwari et al. [180] suggested using Spectrum-based Fault Localisation to obtain the program spectras of S and S', and comparing these program spectras. Disparities between these program spectra can be an indication of a fault in S'. In their approach, S' is essentially the SUT, and S acts as a reference implementation. Thus, their approach can be perceived to be a modified version of N-version Testing, in which program spectra are compared instead of outputs. Since program spectra can represent event flow, this modified version of N-version Testing may be able to test the flow of events. However, there is little evidence to suggest that this approach can generalise outside of regression testing. Thus, we believe that feasibility studies that explore the use of this approach in

other contexts would be valuable.

Let S denote the SUT and S_R be a reference implementation of S, such that S and S_R have faults that result in the same failed outputs, S^o and S_R^o respectively. Since N-version Testing detects a fault by checking $S^o \neq S_R^o$ [116], it cannot detect these faults. This is referred to as a correlated failure. Numerous guidelines for reducing the likelihood of correlated failures are available. The remainder of this section explores these guidelines.

A fault is more likely to be replicated in both S and S_R if the same team develop both, because they might be prone to making the same types of mistakes [137]. Thus, one guideline includes using independent teams for each system [116] e.g. using 3rd party software as S_R. However, this does not eliminate the problem completely because independent development teams can also make the same mistakes [137]. This could be because certain systems are susceptible to particular fault types. Thus, another guideline involves diversifying the systems to reduce the overlap of possible fault types across systems [116]. This can be achieved by using different platforms, algorithms [113], design methods, software architectures, and programming languages [116] for each system. For example, pointer related errors cannot lead to correlated failures if S and S_R are encoded in C++ and Java respectively.

The third guideline we consider revolves around manipulating the test suite. Some test inputs lead to correlated failures, and others do not [18]. Thus, the chance of detecting a fault depends on the ratio of inputs that lead to a correlated failure (CF) to inputs that do not (we refer to non-correlated failures as standard failures (SF)). Since multiple faults may collectively contribute to populating CF and diminishing SF [18], one could adopt a strategy of re-executing the test suite when a fault is removed because this may improve the $CF : SF$ ratio. To demonstrate, let f_1 and f_2 represent two faults in the SUT, and $\{1, 2, 3, 4, 5\}$ be a set of inputs that lead to a correlated failure as a result of f_1. Further suppose that $\{5, 6\}$ is the set of inputs that can detect f_2. Since f_1 causes 5 to lead to a correlated failure, only input 6 can detect f_2; thus by removing f_1, the number of inputs that can be used to detect f_2 doubles, since 5 would no longer lead to a correlated failure.

Although the guidelines discussed above (i.e. using independent development teams, diversifying the systems, and test suite manipulation) can reduce the number of correlated failures, the extent to which they do varies across systems. This is because different systems have outputs with different cardinalities[1], which have been observed to influence the incidence of correlated failures [18].

2.2.2 Usability

The only manual tasks in N-version Testing are procuring RIs and debugging; thus discussions regarding usability will revolve around these issues. This section discusses the former, and the latter is covered in Section 2.2.3.

At its inception, the recommended method of procuring RIs for the purpose of N-version Testing was development [190]. Developing RIs can require substantial time and effort [81]. Many solutions have been proposed, that might reduce the labour intensiveness of this task. For example, Davis and Weyuker [53] recognised that the performance of an RI is not important, because it is not intended to be production quality code. They also realised that programs that are written in High Level Programming Languages have poorer runtime efficiency, but are quicker and easier to develop [53]. To that end, they suggest using such languages for the development of RIs. Similarly, we suspect that

[1] Output cardinality refers to the proportion of inputs that map to an output.

it might be possible to sacrifice other quality attributes, to make RI development faster and easier.

Solutions that can eliminate development effort completely have also been proposed. For example, it has been reported that the previous versions of the same system [206], or other systems that implement the same functionality [24] could be used as RIs. RIs could also be automatically generated e.g. through Testability Transformations [125]. Testability Transformations automatically generate RIs by modifying the original system's source code, O, into a syntactically different version, M, such that M and O are observationally equivalent if O has been implemented correctly, but not if O is faulty. Although the technique is only applicable to a small range of fault types, it has been argued that these faults are widespread [125].

Component Harvesting is an alternative procurement strategy. It involves searching online code repositories with some of the desired RI's syntax and semantics specification [81]. Hummel et al. [81] assert that this substantially reduces procurement effort. However, this may not always be true; other activities may be introduced that will offset effort gains. For example, an RI's relevant functionality may depend on irrelevant functionality; such dependencies must be removed [83]. The SUT and RIs must also have a common input-output structure [37]. Thus, it may be necessary to standardise the structure of the input and output [150]. Atkinson et al. [10] remark that the effectiveness of the search depends on how well the user specifies the search criteria. It is therefore possible for the search to return systems that cannot be used as RIs. Additionally, systems that have unfavourable legal obligations may also be returned [10]; using these systems as RIs may therefore be infeasible. Identifying and removing such systems from the search results may be labour intensive.

Suitable RIs may not exist [137]. This means Component Harvesting may be inapplicable in some cases. Additionally, the applicability of the technique is restricted by its limitation to simple RIs [10] i.e. RIs that are limited in terms of scale and functionality. This means that the technique can only support simple SUTs.

Although Testability Transformations and Component Harvesting can substantially improve the usability of N-version Testing, these techniques clearly have limited generalisability i.e. the former and latter only cater for a limited range of faults and systems respectively. Further research that results in improvements in their generalisability could add significant value. For example, Component Harvesting might be extended to more complex RIs as follows: since the semantics of simple RIs are understood, it may be possible to automatically combine multiple simple RIs into a more complex RI.

2.2.3 Cost

The cost of N-version Testing is a divisive issue. The cost of obtaining RIs is particularly contentious. Many claim that the RI procurement process is expensive because it may involve the re-implementation of the SUT [77]. However, others argue that this process can be inexpensive because it can be automated by procurement strategies like Component Harvesting [81]. However, as discussed in Section 2.2.2, these strategies are only applicable under certain conditions and so manual re-implementation may be necessary in some situations. This means that the cost of obtaining RIs can vary.

In manual testing, the tester must manually verify the output of a test case. In N-version Testing, this process is automated [18]. This means that test execution can be cheaper in N-version Testing in comparison to manual approaches; thus N-version Testing could be cheaper if a large number of

test cases are required. It might be necessary to generate additional test cases because of software maintenance [9]. Thus, the requirement for a larger number of test cases might be correlated with update frequency. Update cost can be exacerbated by N-version Testing because changes may have to be reflected across all RIs [113]. This cost may be further exacerbated, depending on the RI's maintainability [116]. This may offset the cost effectiveness gains obtained from cheaper test cases in some scenarios.

Let $V1$ be the SUT and $R1$ be an RI based on $V1$. Suppose that $V1$ was updated to become $V2$. Some test cases that are applicable for $V1$ (and by implication, $R1$) may also be applicable for $V2$ [206]. Thus, instead of updating $R1$ to be consistent with $V2$, one could simply restrict testing to these test cases. This might alleviate update costs. However, such an approach clearly cannot cater for new functionality [81].

The impact of increasing the number of RIs on cost effectiveness is also unclear. A failure's cost can be high [83, 116], which means substantial cost savings may be obtained by detecting a fault that could result in such a failure. Since the number of RIs is positively correlated with effectiveness [18], the chance of obtaining these cost savings can improve if more RIs are used. However, as mentioned above, the cost of developing an RI can be expensive [113]. Thus, increasing the number of RIs will inherently increase development cost. Clearly, the direction of the correlation between the number of RIs and cost is dependent on whether a sufficiently expensive fault is found.

It is unclear which system is the source of failure [37]; this means that one must debug multiple systems. Thus, using more RIs can lead to an increase in debugging costs. However, using multiple RIs enables the establishment of a voting system, where each RI (and the SUT) votes for its output [146]. Systems that are outnumbered in a vote are likely to be incorrect. Thus, debugging effort can be directed and therefore minimised. Unfortunately, correct systems can be outnumbered in the vote [83]; therefore a voting system may have limited impact in some situations.

2.2.4 Content-based Image Retrieval

Some systems produce graphical outputs. The correctness of graphical outputs can be verified by comparing them to reference images [54]. Reference images could be obtained from RIs. Oliveira et al. [147] proposed combining a Content-based Image Retrieval System with feature extractors and similarity functions to enable the automated comparison of such outputs with reference images, based on their critical characteristics.

Unfortunately, the application of their technique is not fully automated. For example, one must acquire appropriate feature extractors and similarity functions [147]. However, some of this manual effort may not always be necessary. For instance, many feature extractors and similarity functions are freely available [147], thus one may not have to develop these, if these free ones are appropriate.

2.3 Metamorphic Testing

In Metamorphic Testing (MT), a set of test cases, called the Metamorphic Test Group (MTG), is generated. MTG has two types of test cases. Source test cases are arbitrary and can be generated by any test case generation strategy [35, 71], whilst follow up test cases are generated based on specific source test cases and a Metamorphic Property [135]. A Metamorphic Property is an expected

relationship between source and follow up test cases.

For example, consider a self-service checkout that allows a customer to scan product barcodes and automatically calculates the total price. The Metamorphic Property might state that a shopping cart that consists of two instances of the same product type should cost more than a shopping cart with just one. Let B_1 and B_2 be instances of the same product, and $SC_1 = \{B_1, B_2\}$ denote a shopping cart containing both instances. Suppose that SC_1 is a source test case. Based on this Metamorphic Property and source test case, MT might use subset selection to derive two follow up test cases: $SC_2 = \{B_1\}$ and $SC_3 = \{B_2\}$. Thus, the MTG may consist of SC_1, SC_2, and SC_3. The Metamorphic Property in conjunction with the MTG is called a Metamorphic Relation (MR). MRs are evaluated by executing the MTG and checking that the Metamorphic Property holds [93] between these executions; in this case checking that the price of SC_1 is greater than the price of SC_2, and the price of SC_1 is greater than the price of SC_3.

A permutation relation is an MR where changes in the input order has a predictable effect on the output. For example, consider a sort function, $Sort(I)$, where I is a list of integers. A permutation relation might develop the following source and follow up test cases: $Sort(1, 3, 2)$ and $Sort(3, 1, 2)$, with the expectation that their outputs are the same. Some refer to MT as Symmetric Testing in situations where only permutation relations are used [68].

Like N-version Testing, MT was created to alleviate the oracle problem. In particular, MT attempts to resolve oracle problems where the test outcome is unpredictable due to a lack of prior knowledge. As has been made apparent above, MT does not rely on predicted test outcomes to verify the correctness of the SUT. Thus, MT can operate in the presence of this oracle problem. MT has also been shown to be effective for a large range of different oracle problems, including complex [71] (i.e. systems that involve non-trivial processing operations) and data intensive systems [37], because the process of evaluating an MR can be inexpensive.

2.3.1 Effectiveness

Experiments on MT's effectiveness have produced varied results, ranging from 5% [137] to 100% mutation scores [170]. Several factors, that influence effectiveness and thus may explain this disparity, have been reported. These factors can broadly be categorised as follows: coverage [102], characteristics, the problem domain, and faults. This section explores these factors. For generalisability purposes, our discussions are limited to implementation independent issues.

2.3.1.1 Coverage

Numerous strategies for maximising the coverage of MT are available. For example, it has been observed that some MRs place restrictions on source test cases [101]. Thus, one's choice of MRs could constrain a test suite's coverage. Coverage could be maximised by limiting the usage of such MRs.

Núñez and Hierons [141] observed that certain MRs target specific areas of the SUT. This means that increasing the number of MRs that are used, such that the additional MRs focus on areas of the SUT that are not checked by other MRs, could increase coverage. Merkel et al. [126] state that since testing resources are finite, there is a trade-off between the number of MRs and test cases that can be used. Therefore, increasing the number of MRs to implement the above strategy could limit the test suite size. Thus, the aforementioned coverage gains could be offset. The optimal trade-off is context

dependent.

Let P be a program consisting of three paths $P = \{\{s_1, s_2\}, \{s_2\}, \{s_2, s_3\}\}$, and let MR_1 and MR_2 be MRs that each have an MTG that consists of two test cases. Suppose that MR_1's MTG covers the first and second path and thereby executes statements s_1 and s_2, and that MR_2's MTG covers the first and third path and so covers all three statements. This demonstrates that an MR's MTG can obtain greater coverage, if the paths that are traversed by each of its test cases are different [20]. Several guidelines have been proposed to design MRs to have such MTGs. For example, white box analysis techniques [58], or coverage information generated by regression testing [20] could assist in the identification of MRs that have MTGs with different test cases. MRs that use a similar strategy to the SUT tend to have MTGs that have similar source and follow up test cases [124], and thus should be avoided. Different MRs can have different MTG sizes [20]. It seems intuitive that MRs that have MTGs that consist of a larger number of test cases are more likely to have test cases that traverse dissimilar paths.

2.3.1.2 Characteristics

An MR has many characteristics that can be manipulated to improve its effectiveness. For example, it has been observed that decreasing the level of abstraction of an MR can improve its fault detection capabilities [88]. This section explores these characteristics and their relationships with effectiveness.

MRs can vary in terms of granularity e.g. application or function level. In a study conducted by Murphy et al. [134], it can be observed that MRs that are defined at the application level can detect more faults than MRs that are defined at the function level, in some systems. This means that MRs that were defined at a higher level of granularity were more effective for these systems. Interestingly, the converse was also observed for other systems [134], and so the most effective level of granularity might depend on the system. Regardless, both MR types found different faults [134], and thus, both can add value in the same context.

It has been reported that an MR that captures a large amount of the semantics of the SUT (i.e. an MR that reflects the behaviours of the SUT to a greater degree of completeness and accuracy) can be highly effective [124]. Let MR_r and MR_p be two MRs, such that MR_r captures more of the semantics of the SUT than MR_p. This suggests that MR_r might be more effective than MR_p. We believe that certain test cases can capture some of the semantics of the SUT. Let tc be such a test case. It may therefore be possible for MR_p to obtain a comparable level of effectiveness to MR_r, if MR_p is evaluated based on tc, because the additional semantics in tc may counteract the deficit of such semantics in MR_p. However, recall that some MRs place restrictions on test inputs [101]; this may limit the scope for using test cases like tc with MRs like MR_p.

The fourth widely reported characteristic is strength. Let MR_1 and MR_2 be two MRs, such that MR_1 is theoretically stronger than MR_2. This means that if one can confirm that MR_1 holds with respect to the entire input domain, then this implies that MR_2 also holds with respect to the entire input domain [124]. This implies that MR_1 can detect all of the faults that can be detected by MR_2, in addition to other faults. Some regard MRs like MR_2 to be redundant [176]. Interestingly, a study conducted by Chen et al. [38] compared the failure detection rate[2] of 9 MRs. The weakest MR obtained the highest failure detection rate for 15/16 of the faults, whilst the strongest MR obtained the

[2]The failure detection rate measures the proportion of test cases that detect a fault.

lowest failure detection rate for 13/16 faults. This suggests that strong MRs are not necessarily more effective than weak MRs [38], and weak MRs are therefore not redundant. Mayer and Guderlei [124] realised that weak MRs can have more failure revealing test cases than stronger MRs. This may explain why weak MRs can be more effective.

Black box MT emphasises the development of strong MRs. It is therefore not surprising that the observation that weak MRs can be more effective than strong MRs led Chen et al. [38] to conclude that black box MT should be abandoned. Proponents of this argument view an understanding of the algorithm structure as necessary [38]. Although this argument has a strong theoretical foundation, Mayer and Guderlei [124] have questioned the practicality of the position. One must consider all input-output pairs to deduce the relative strength of one MR to another, which can be infeasible in practice [124]. Thus, opponents contend that categorising MRs based on their strength is impractical, and by implication, deciding to abandon black box MT based solely on MR strength is nonsensical [124]. While we agree with Mayer and Guderlei [124] that it may be impractical to determine whether one MR is stronger than another, we disagree with the notion that this threatens the validity of the argument of Chen et al. [38], since knowledge about the relative strength of two MRs is not necessary to leverage the advice of Chen et al. [38].

Tightness is another major characteristic [112]; tighter MRs have a more precise definition of correctness. For example, a tight MR may check $X == (Y \times 2)$; only one answer is acceptable. A looser MR may check $(X > 2)$; while $(X \leq 2)$ indicates a fault, an infinite number of answers are acceptable. Therefore, tighter MRs are more likely to be effective [126]. Although tight MRs are preferable, they may be unavailable. For example, consider a non-deterministic system that returns a random output that is approximately two times larger than the input. A tight MR is not available because predicting the precise output is impossible, however, the following loose MR can be used: $output < (input \times 4)$ [138].

Another important characteristic is the soundness of an MR. A sound MR is one that is expected to hold for all input values. Conversely an MR that is unsound is only expected to hold for a subset of the input values [133]. Unlike sound MRs, unsound MRs are prone to producing false positives[3] [133]. It might be advisable to avoid using such MRs, to curtail false positives. However, it has been reported that MRs that are less sound might be capable of detecting faults that cannot be detected by MRs that are more sound [133]. Thus, such MRs might add value.

2.3.1.3 Problem domain

It has been reported that MT is more effective when one uses multiple MRs, instead of just one MR [126]. Since MRs are domain specific [37], the total number of potential MRs in one problem domain can be different than in another. For example, Chen et al. [38] found nine MRs for Dijkstra's Algorithm, whilst Guderlei and Mayer [70] could only find one MR for the inverse cumulative distribution function. Therefore, the problem domain is likely to directly influence MT's effectiveness.

Specialised variants of MT have been developed to account for the characteristics of certain problem domains. For example, Murphy et al. [137] propose Metamorphic Heuristic Oracles to account for floating point inaccuracies and non-determinism. This approach involves allowing MT to interpret

[3]In the context of software testing, a testing technique is said to have reported a false positive if it incorrectly reports a failure.

values that are similar, as equal [133]. The definition of “similar” is context dependent [137], thus general guidance is limited.

2.3.1.4 Faults

MT and its variants can detect a diverse range of faults e.g. MT can find faults in the configuration parameters [141] and specifications [37], and Statistical Metamorphic Testing (see Section 2.6.3) can find faults that can only be detected by inspecting multiple executions [136]. However, MRs are necessary, but not sufficient [46]; they are not effective for all fault types e.g. coincidentally correct faults [20, 202].

Specifications can be used as a source of inspiration for the MR identification process [88, 109]. It has been reported that the effectiveness of MT can be compromised by errors in the specification [109]. This could be because errors in the specification may propagate to the MRs, if the MRs have been designed based on the specification. The same specification errors may have also propagated into the SUT, thus there might be scope for correlated failures (see Section 2.2.1). This might explain why MT cannot find certain faults. One might reduce this risk by using other sources of inspiration e.g. domain knowledge [37] or the implementation [135].

2.3.2 Usability

2.3.2.1 Prerequisite Skills and Knowledge

Mishra et al. [129] observed that students performed better on class assignments revolving around equivalence partitions and boundary value analysis, when compared to MT. This suggests that MT might be more difficult to grasp than other testing techniques. This could be because MT requires a wide skillset to operate.

Poon et al. [155] claim that MR implementation requires limited technical expertise. However, others have stated that the tester might not be competent enough to implement MRs [25], which indicates that developing MRs might be difficult. These conflicting conclusions suggest that the difficulty of MR development might be context dependent.

One’s domain expectations might not necessarily match the implementation details of the SUT. This disparity might be a result of an intended design decision [131]. For example, the SUT’s precision may be compromised in favour of efficiency. Thus, if one is not aware of such design decisions and design MRs purely based on domain expectations, the MR might erroneously interpret this disparity as a failure. Thus, knowledge about the implementation details of the SUT might be important.

Domain experts can identify more MRs, that are more effective, more productively than non-domain experts [41]. This suggests that domain knowledge is also important. Therefore, if one lacks adequate domain knowledge, it is advisable to consult domain experts [110]. MRs are identified in booms and slumps; the SUT is investigated during a slump to develop new intuitions that can be used to identify MRs, and such MRs are defined in boom periods [45]. This iterative process affords further opportunities to continuously supplement one’s domain knowledge.

An experiment conducted by Zhang et al. [206] found that different developers can identify different MRs. This is not surprising because different people have different domain knowledge. It may therefore be advisable to leverage a team [155], because this may ensure greater coverage over the domain

knowledge. A small team e.g. consisting of 3 people has been shown to be sufficient [109].

2.3.2.2 Effort

A number of factors affect the effort required to apply MT. For example, it has been observed that an MR that has been developed for one system, might be reusable in another system [102]. Thus, MT might be easier to apply in situations in which MRs that were developed for other systems are available. Another example is MTG size. Since it is not apparent which test case in the MTG contains the failure, all test cases must be considered during debugging [25]. This means that effort can be substantially reduced if the MTG size is reduced. Alternatively, Liu et al. [112] proposed a method that could provide some indication of the likelihood that a particular test case in MTG executed the fault. Their method deems a test case to be more likely to have executed the fault, if it was executed by more violated MRs. This could be used to direct debugging effort. Another alternative is Semi-Proving, which is covered in Section 2.3.6.

The most significant factor affecting effort is believed to be the difficulty of MR identification. Thus, most research has been conducted on this factor. For example, Chen et al. [45] found that MR identification is difficult because inputs and outputs must be considered simultaneously [45]. They alleviated this by automating input analysis, thereby constraining the tester's attention to outputs [45]. The technique specifies a set of characteristics, called "Categories"; each is associated with inputs that manipulate it. These inputs are subdivided into "choices"; all inputs belonging to a particular choice manipulate the characteristic in the same way.

A test frame is a set of constraints that define a test case scenario. Pairs of test frames (that correspond to source and follow up test cases) can be automatically generated by grouping various categories and choices together, such that they are "Distinct" and "Relevant" (i.e. marginally different). For example, let $Function(a, b, c, x, y, z)$ be a function with 6 input variables; a distinct and relevant pair may only differ by one of these variables e.g. z. These test frames produce test cases that are executed to obtain a set of outputs, which can be manually checked for relationships. The process of automatically generating test frames and manually analysing them is iterative [45]; since an infeasible number of pairs typically exist, the terminating condition is the tester's satisfaction with the identified pool of MRs [45].

The empirical evidence is promising; people's performance with respect to MR identification improved, and novices achieved a comparable level of performance to experts [45]. However, the technique has an important limitation; it can currently only support MRs that are composed of one source and follow up test case [45].

Kanewala and Biemen [93] alternatively propose training Machine Learning classifiers to recognise operation sequence patterns that are correlated with particular MRs. Such a classifier can predict whether unseen code exhibits a particular MR. Results have been promising; the technique has a low false positive rate, and can identify MRs even when faults are present in the SUT [93].

Although this technique achieves greater automation [93] than the approach devised by Chen et al. [45], additional human involvement is introduced elsewhere. For example, training datasets are necessary for the machine learning classifiers [95], and obtaining these can be difficult [45]. One may wish to extend the classifier with a graph kernel[4], that has parameters [95] that might have to be

[4] A graph kernel is a function that compares graphs based on their substructures.

tuned to improve accuracy. Furthermore, since each classifier is associated with one MR type [93], these additional tasks must be repeated for each MR type.

2.3.3 Efficiency

There is a time cost associated with test case generation and execution [31]. As discussed in Section 2.3.1.1, different MRs have different MTG sizes. This means that some MRs might incur greater time costs than others. Thus, one might improve the efficiency of MT by restricting oneself to MRs with smaller MTGs. However, as was discussed in Section 2.3.1.1, MRs with larger MTGs might obtain greater coverage, thus such a restriction might reduce the effectiveness of the technique. Alternatively, one might consider using parallel processing — the test cases in the MTG can be executed simultaneously [133].

Other approaches include combining MRs in various ways to make more efficient use of test cases. For example, one could use the same test cases for different MRs [31]. One method of implementing such an approach might be Iterative Metamorphic Testing. This involves combining MRs together, $\langle MR_i, MR_{i+1}, MR_{i+2}...MR_n\rangle$, such that the follow up test case(s) of MR_i are used as the source test case(s) of MR_{i+1} [192]. Combination relations is another possible method.

2.3.4 Combination Relations

Liu et al. [110] suggested defining a new MR that is composed of multiple MRs. For ease of reference, we called such an MR a "combination relation". By evaluating this single MR, one implicitly evaluates all of the constituent MRs, and thus makes more efficient use of test cases. Logic dictates that a single MR that embodies multiple MRs would have a level of effectiveness that is equivalent to the sum of its constituent parts [110]. Interestingly however, it has been found that such an MR can actually obtain a higher level of effectiveness than its constituent MRs [110]. This could be because one MR in the combination relation may empower another. For example, MRs MR_n and MR_c may be effective for numerical and control flow faults respectively; combining the two may extend MR_n's capability to control flow faults.

Conversely, effectiveness can deteriorate; Liu et al. [110] observed that including a loose MR in a combination relation can reduce the combination relation's overall effectiveness. Thus, they advocate only combining MRs that have similar tightness. They also observed that some MRs might "cancel" out other MRs either partially or completely [110]. This may also explain why the effectiveness of a combination relation might deteriorate. These observations suggest that one may be limited in one's choice regarding which MRs can be combined, and by implication, the technique might be inapplicable in some scenarios [130].

Different MRs can accommodate different subsets of the input domain [59]. Since an input must be suitable for all MRs in the combination relation, additional restrictions might have to be placed on constituent MRs. For example, suppose that MR_1 and MR_2 are two MRs in a combination relation. MR_1 can accommodate five test inputs, $\{t_a, t_b, t_c, t_d, t_e\}$, and MR_2 can only accommodate three test inputs, $\{t_a, t_b, t_f\}$. In this situation, it is not possible to use test cases $\{t_c, t_d, t_e, t_f\}$. This means that MRs that can accommodate larger subsets may be more useful [59]. This could also explain why a combination relation's effectiveness might deteriorate.

2.3.5 Metamorphic Runtime Checking

In Metamorphic Runtime Checking, MRs are instrumented in the SUT, and evaluated during the SUT's execution. One of the benefits of this approach is that MRs are evaluated in the context of the entire SUT [134]. This can improve the effectiveness of MT. To illustrate, Murphy et al. [134] observed that MRs that are evaluated in one area of the system, could detect faults in other areas.

Unfortunately, unintended side effects can be introduced during instrumentation [134]. For example, consider a function, $F(x)$, and a global counter variable, I. I is incremented every time $F(x)$ is executed. A follow up test case that executes $F(x)$ will inadvertently affect I's state. Thus, sandboxing may be advisable [133].

Sandboxes introduce additional performance overheads [132]. However, since Metamorphic Runtime Checking uses test data from the live system [138], the generation of a source test case is no longer necessary. These efficiency gains may offset the losses from the performance overheads incurred from sandboxes.

To improve the efficiency of the approach further, some have suggested parallel execution [133]. It has been observed that the number of times each MR is evaluated is dependent on the number of times each function is invoked [134]. To illustrate, let $f_1()$ and $f_2()$ be two functions in the same system, such that $f_1()$ is always invoked twice as many times as $f_2()$ because of the control flow of the system. Suppose that MRs MR_1 and MR_2 are evaluated each time $f_1()$ and $f_2()$ are invoked respectively. Since MR_1 is evaluated twice as many times as MR_2, MR_1 would add more performance overheads than MR_2. Thus, one could prioritise MRs like MR_2 over MRs like MR_1 to improve performance.

2.3.6 Semi-Proving

An MR's verdict only indicates the SUT's correctness for one input. Semi-Proving attempts to use symbolic execution to enable such a verdict to generalise to all inputs [47].

In Semi-Proving, each member of an MR's MTG, $MetTestGrp = \{tc_1, tc_2, ...tc_n\}$, is expressed, using symbolic inputs, as constraints that represent multiple concrete test cases. Each test case, tc_i, in MTG is symbolically executed, resulting in a set of symbolic outputs, $O_i = \{o_{ij}, o_{ij+1}, ...o_{in}\}$, and corresponding symbolic constraints, $C_i = \{c_{ij}, c_{ij+1}, ...c_{in}\}$, that the output is predicated on. Let CP be the Cartesian product of each C_i i.e. $C_1 \prod C_2 \prod ... \prod C_n$. For each combination $comb = \langle C_{1a}, C_{2b}, ...C_{nc}\rangle$ in CP, the conjunction of all members of *comb* should either result in a contradiction or agreement. For each agreement, Semi-Proving checks whether the MR is satisfied or violated under the conditions represented by *comb*.

Since all concrete executions represented by a symbolic execution are accounted for, it is possible to prove the correctness for the entire input domain, with respect to a certain property [47]. However, this might not always be feasible. For example, in some systems, certain loops, arrays, or pointers could cause such a large number of potential paths to exist, that it would be infeasible for Semi-Proving to check them all exhaustively [47]. To alleviate this problem, one could restrict the application of the technique to specific program paths, replace some symbolic inputs with concrete values, use summaries of some of the SUT's functions instead of the functions themselves, or restrict the technique with upper-bounds [47]. Chen et al. [47] realised that the correctness of some symbolic test cases can be inferred from others. For example, consider the max function and the following two symbolic test cases: $max(x, y)$ and $max(y, x)$. Since these test cases are equivalent, only one must be executed

to deduce the correctness of both. Optimising resource utilisation through this strategy may also alleviate the problem.

Obtaining such coverage can improve the fault detection effectiveness of MT [47]. Improvements in effectiveness for subtle faults e.g. missing path faults, has been reported to be particularly noteworthy by several researchers [47, 69]. Another advantage of greater coverage is improvements in debugging information. In particular, there is greater scope for the precise failure causing conditions [47] and test cases [112] to be identified. Whether this improves debugging productivity is questionable though; investigating this information requires manual inspection of multiple (possibly all) execution paths [47].

2.3.7 Heuristic Test Oracles

Heuristic Test Oracles are a loose variant of Metamorphic Testing. In this approach, expected input-output relationships are initially identified e.g. input increase implies output decrease. The SUT is then executed multiple times with different inputs, to obtain a set of outputs. These inputs and outputs are used in conjunction with each other to check whether the expected input-output relationship holds [77].

Thus, Heuristic Test Oracles can only be applied to systems that have predictable relationships between inputs and outputs [77]. Some systems may not have relationships that span the entire input domain. In such situations, it might be possible to define heuristics for a subset of the input domain [77]. For example, Sine's input domain can be split into three subdomains: *Subdomain One* = $\{0 \leq i \leq 90\}$, *Subdomain Two* = $\{90 \leq i \leq 270\}$, and *Subdomain Three* = $\{270 \leq i \leq 360\}$. A positive correlation between the input and output can be observed in *Subdomain One* and *Subdomain Three*, whilst a negative correlation is assumed in *Subdomain Two* [77].

It has been reported that these oracles are effective [113]. Some have also claimed that these oracles are faster and easier to develop [77] and maintain [113] than N-version Testing based oracles. Heuristic Test Oracles also have high reuse potential [77], thus implementation may be bypassed completely in some cases.

2.4 Assertions

Assertions are Boolean expressions that are directly embedded into the SUT's source code [12]. These Boolean expressions are based on the SUT's state variables e.g. $X > 5$, where X is a state variable. Assertions are checked during the execution of the SUT, and may either evaluate to true or false; false indicates that the SUT is faulty [74]. Our general discussions on Assertions in this section are based on the above definition. We are aware that some people use alternative definitions, for example, some definitions allow one to augment the SUT e.g. by introducing auxiliary variables (see Section 2.4.1). Our discussions regarding such alternative definitions of the technique will be clearly indicated in the text.

Unlike N-version Testing and Metamorphic Testing, Assertions were not originally designed to alleviate the oracle problem. However, it has been observed that in order to evaluate an assertion, one does not have to predict the test outcome [12]. This means that assertions are applicable to certain classes of oracle problem e.g. for situations in which it is not possible to predict the test outcome.

2.4.1 Effectiveness

Several characteristics of Assertions have been found to influence effectiveness. For example, one characteristic is that Assertions must be embedded in source code [174]. Unfortunately, this can cause unintended side effects that manifest false positives e.g. additional overheads [94] may cause premature time-outs. Thus, one must carefully write assertions to avoid side effects [132].

Assertions can be written in independent programming or specification languages e.g. assertions can be written in Anna, and be instrumented in a program written in Ada [12]. Some languages are particularly intuitive for certain tasks e.g. LISP for list manipulation. One could exploit these observations, by writing Assertions in the most apposite language for the types of tasks to be performed. This might reduce the chance of introducing unintended side effects. Unfortunately, this approach can also increase the chance of introducing unintended side effects if it causes deterioration in readability. One can use polymorphism; assertions can be specified in a parent class, and a child class can inherit assertions from the parent class [5]. By using such a strategy, one can isolate assertions (in parent classes) from the system's source code (in child classes); this might alleviate readability issues.

The code coverage of Assertions can be limited, depending on the nature of the program. For example, let *List* be an array. To test *List*, an Assertion may assert that some property holds for all members of *List*. It may be infeasible to evaluate this Assertion, if *List* has an large number of elements [12]. Thus, it may be infeasible for assertions to be used in areas of the code that have large arrays. Consider another example; Assertions can only check a limited range of properties that are expected to hold at particular points in the program e.g. $Age \geq 0$ [74]. This means their coverage could be limited. According to some alternative definitions of Assertions, auxiliary variables can be introduced into the system, for the purpose of defining Assertions [12]. Introducing auxiliary variables might create new properties that can be checked by Assertions, and thus alleviate the problem. For example, suppose that x is a variable in the system, and y is a newly introduced auxiliary variable; we might include an assertion such as $x > y$.

Another facet of coverage is oracle information. One aspect of oracle information is the types of the properties that can be checked by a technique. For example, Assertions can check the characteristics of the output or a variable e.g. range checks [174], or how variables might be related to one another [92] e.g. $X \neq Y$. This makes Assertions particularly effective for faults that compromise the integrity of data that is assigned to variables [133]. Another aspect of oracle information is the number of executions that test data is drawn from. Test data from multiple executions is necessary for certain faults e.g. the output distributions of a probabilistic algorithm [136]. Assertions are unable to detect such faults because they are restricted to one execution [136].

Some believe that Assertions can be used to detect coincidentally correct faults [92]. This supposition probably stems from the fact that Assertions have access to internal state information, and thus could detect failures in internal states, that do not propagate to the output. To the best of our knowledge, there isn't any significant empirical evidence that demonstrates that Assertions can cope with coincidental correctness. Thus, investigating this might be a useful future research direction.

It has been observed that the detection of some coincidentally correct faults may require oracle information from multiple states (see Section 3.3.6.1). Based on an alternative definition of Assertions, Baresi and Young [12] report that Assertions can check multiple states, if they are used in conjunction with state caching. However, they also remark that state caching may be infeasible, if a large amount

of data must be cached. In such situations, Assertions cannot detect such coincidentally correct faults. Additionally, they observed that Assertions cannot correlate events between two modules that do not share a direct interface. This means that assertions may not be able to check certain states simultaneously, and thus may render it incapable of detecting certain coincidentally correct faults. These observations demonstrate that, despite the fact that assertions have access to internal state information, they may not necessarily be effective for coincidental correctness, even when state caching is feasible.

MT has access to information from multiple executions. Sim et al. [174] combined MT and Assertions, such that Assertions are evaluated during the execution of a Metamorphic Test's source and follow up test cases. This integration may alleviate some of the oracle information coverage issues described above.

2.4.2 Usability

One key skill that is a part of many developers repertoires is program comprehension i.e. the capability to understand the logic of a program by inspecting the source code [206]. Developers have experience with modifying source code [206] e.g. to add new functionality. Therefore, developers will be comfortable with comprehending and modifying the system's source code. These tasks are integral to the application of assertions. This led Zhang et al. [206] to conclude that constructing assertions can be more natural than developing oracles from other approaches like Metamorphic Testing. However, Assertions assumes that the tester has knowledge about the problem domain, or the SUT's implementation details [92]. This means that an assertion could require more effort to construct in situations in which the tester has limited knowledge regarding these areas, since they would have to first acquire this knowledge. Other factors that affect the effort required to construct an assertion include the level of detail the assertion is specified at [5] and the programming language's expressiveness [139].

Some tools can support the development of assertions e.g. the assert keyword in some programming languages [12], and invariant detection tools. Invariant detection tools can be used to automatically generate assertions. They work by conducting multiple executions and recording consistent conditions [92]; these conditions are assumed to be invariant and so pertain to assertions. It is typically infeasible to consider all executions; thus only a subset is used. Variant conditions may be consistent across this subset, and thus may be misinterpreted as invariant. Thus, invariant detection tools can produce spurious assertions [134]. Therefore, the manual inspection of suggestions from these tools is necessary [92]. Unfortunately, manual inspection can be error prone; cases have been observed where 50% of the incorrect invariants that were proposed by such a tool were misclassified by the manual inspection process [74].

2.4.3 Multithreaded Programs

Interference in multi-threaded environments can cause assertions to produce false positives [5]. Several guidelines have been proposed to circumvent this. Firstly, assertions can be configured to evaluate under safe conditions e.g. when access to all required data has been locked by the thread [5]. Secondly, the application of assertions can be restricted to blocks of code that are free from interference [5]. Recall that assertions can add performance overheads. This is problematic in multi-threaded environments, because these performance overheads can introduce new or remove important interleavings.

This can be alleviated by load balancing [5].

2.4.4 Further discussion

Research on assertions in the context of the oracle problem is scarce. Most studies either combine it with other techniques or use it as a benchmark. This implies that Assertions are assumed to be at least moderately effective for non-testable programs; but this is largely unsubstantiated. Thus, empirical studies that test this assumption may be valuable.

The literature reported in this Mapping Study did not present guidelines for assertion use in non-testable systems. We therefore believe that future work that establishes such guidelines in the context of the oracle problem will be valuable.

2.5 Machine Learning

Machine Learning (ML) Oracle approaches leverage ML algorithms, in different ways, for testing purposes. One method involves training a machine learning algorithm, on a training dataset, to identify patterns that are correlated with failure. The SUT can be executed with a test case, and this trained machine learning algorithm can then be used to check for such patterns in this test case execution. For example, Chan et al. [24] constructed a training dataset, in which each data item corresponded to an individual test case, and consisted of a set of features that characterised the input and output of this test case. Each data item was also marked as "passed" or "failed". A classifier was trained on this training dataset, and so became capable of classifying test cases, that were executed by the SUT, as either passed or failed. Another method involves training a machine learning algorithm to be a model of the SUT; thus, the ML algorithm becomes akin to a reference implementation in N-version Testing [146].

ML techniques were not originally developed for testing non-testable programs, but they can be applied to such programs [92]. To illustrate, ML Oracles draw their oracle information from training datasets, which can be obtained when information about the expected test outcome is not available prior execution. This can allow them to test systems for which the expected test outcome is not known before the execution.

2.5.1 Effectiveness

2.5.1.1 Design and application of ML Oracles

Several factors affect the effectiveness of ML Oracles. The first set of factors concerns the composition of the training dataset. It has been reported that the balance of passed and failed test cases can affect bias [24]. Datasets can also vary in terms of size. Larger datasets have less bias [24], and are less susceptible to the negative effects of noise [64].

A training dataset must often be reduced to a set of features that characterise it, because the form of the training dataset is seldom appropriate for ML. This is typically achieved by using one or more feature extractors. The second set of factors revolves around the number of feature extractors one uses. Several trends between the number of feature extractors used and effectiveness can be observed: improvement, stagnation, and decline. Two of the feature extractors used in a study conducted by

Frounchi et al. [64] include the Tanimoto Coefficient TC and Scalable ODI $SODI$. In this study, it was observed that one set of features that consisted of $\{TC\}$ was the most effective set for negative classifications, and that another set of feature extractors that contained $\{TC, SODI\}$ was the most effective set for positive classifications. Clearly, the addition of $SODI$ to a set of feature extractors that just contains TC can lead to an increase in the accuracy for one type of classification, but a decrease for another type of classification. This implies that an ML Oracle's overall effectiveness can be improved or reduced by adding additional feature extractors, if the improvement in one classification type more than offsets, or is more than offset by the loss of accuracy for other classification types respectively. These implications might explain why one may observe the improvement and decline trends.

Let $G = \{fe_1, fe_2, ...fe_j\}$ be a group of feature extractors, such that all $fe_i \in G$ are highly correlated with one another. Using multiple members from G is unlikely to significantly improve classification accuracy [64]. This could explain stagnation trends. This suggests that one should limit the number of members of G, that are used by ML Oracles. Different feature extractors may have different quality attributes e.g. levels of efficiency [64] or generalisability and so some may be more favourable than others in certain contexts. Thus, one may consider choosing a subset of G based on the quality attributes offered by the different feature extractors in G. Techniques like wrappers and filters can identify and remove feature extractors that will not significantly improve classification accuracy [64], and thus can purge excess members of such a group.

Naturally, one would expect that one major factor that might affect effectiveness, is the choice of ML algorithm. However, it has been reported that the choice of algorithm does not have a significant impact on effectiveness, and thus these algorithms might be interchangeable [64].

2.5.1.2 Limitations

ML Oracles have several limitations, and to the best of our knowledge, these limitations have not been resolved by the community yet. Recall that ML Oracles either predict the output of the SUT and then compare this prediction to the SUT's output, or they classify the output of the SUT as correct or incorrect. This means that such oracles are fundamentally used for black-box testing. It's therefore not surprising that examples of these oracles cannot test event flow [139]. For similar reasons, such oracles would be hindered by coincidental correctness. Some have also observed that the negative impact of coincidental correctness on ML Oracles can be exacerbated, if the ML Oracle is trained based on features of the internal program structure [92, 26]. It has also been reported that some variants of ML Oracles are incapable of testing non-deterministic systems or streams of events e.g. ML oracles based on Neural Networks [139]. We believe that resolving these limitations might be useful avenues for future work.

2.5.2 Usability

2.5.2.1 Design and application of ML Oracles

The previous section revealed that in order to leverage ML Oracles, one must obtain appropriate training datasets, an ML algorithm, and apposite feature extractors. This section explores the user-friendliness of these activities.

We begin by considering training dataset procurement. One approach might include obtaining

an RI of the SUT, and then generating the training dataset from this RI [23]. RIs have several characteristics that influence dataset quality e.g. the correctness of the RI. To illustrate, an RI might have a fault, which means that some of the training samples in the dataset may characterise incorrect behaviours (i.e. failures that manifested from this fault), but be marked as correct behaviours. This reduction in dataset quality can limit the effectiveness of an ML Oracle. For example, the SUT might have the same fault as the RI [92], and this can lead to correlated failures. In addition, it has been observed that the extent to which an RI is similar to the SUT is correlated with accuracy [24], and that oracles based on similar RIs can be accurate, effective and robust [92]. These discussions reveal that one must consider a large number of factors during the RI selection process, which could be difficult.

The nature of the data in the training dataset could also have an impact on effort. As discussed above, one aspect of a dataset's composition is test suite balance (in terms of the proportion of passed to failed test cases). If one's dataset is imbalanced, it may be necessary to expend additional effort to obtain additional data to supplement and balance the dataset. The output of an RI characterises correct behaviours, and the output of mutants of an RI characterise incorrect behaviours [24]. Thus, if one lacks passed test cases, one could execute an RI with test cases, and if one lacks failed test cases, one could execute failure revealing test cases over a set of mutants of an RI. However, one may have to construct raw datasets manually, if a suitable RI does not exist (see Section 2.2.2).

The nature of the input and output data used and produced by an ML algorithm can differ from that of the SUT. Thus, it could be necessary to translate inputs that are used by the SUT into a form that is compatible with the ML algorithm, and to translate outputs into a form that is amenable for comparison with the SUT's output [154]. If such translations are necessary, the developers of ML Oracles must either write additional programs to automate this translation task, or perform the translation task manually.

Experts may have to manually label each training sample in the dataset, if one uses a supervised machine learning algorithm to train one's ML Oracle. An example of this can be found in the work conducted by Frounchi et al. [64]. This obviously means that larger datasets will require substantially more effort to prepare, in these situations. If multiple experts are used, then there is scope for disagreement [64]. The resolution of these disagreements will also add to the overall effort required to apply the technique.

We finally consider feature extractor selection. One's choice of feature extractors is an important determinant of the effectiveness of ML Oracles. For this reason, many believe that a domain expert should be involved in this process [92]. If the developer is not a domain expert, consultation may be necessary.

2.5.2.2 Debugging

ML Oracles can report false positives [24], which means testers may waste time investigating phantom faults. ML Oracles can also produce false negatives [92]. False negatives introduce a delay, which means they can also waste resources. Some ML Oracles have tuneable thresholds. Modifying these thresholds can influence the incidence of false positives and negatives [139], which can enable management of such classification errors. Unfortunately, the optimal threshold values vary across systems [139].

2.5.3 Metamorphic Machine Learning

Metamorphic Machine Learning merges MT and ML, such that an ML Oracle evaluates each member of the MTG, before they are used to evaluate the MR. The integration of MT with ML has been found to improve the effectiveness of ML [26]. However, the level of this improvement depends on the quality of the ML Oracle. To illustrate, Chan et al. [26] observed that the extent of the improvement for ML Oracles that used more training data (and were therefore of higher quality) was lower. They rationalised that this was because there was less scope for MT to offer an improvement. Since ML can detect a fault before all of the test cases in the MTG have been executed [26], one could argue that the union of MT and ML can also enhance the efficiency of MT, because it may not be necessary to execute all test cases to detect a fault.

2.6 Statistical Hypothesis Testing

In Statistical Hypothesis Testing (SHT), the SUT is executed multiple times to obtain numerous outputs, which are aggregated using summary statistics e.g. mean and variance. These aggregated values characterise the distribution of this set of outputs, and are compared (using a statistical test e.g. Mann-Whitney U) to values that delineate the expected distribution. Comparisons that do not yield significant differences can be interpreted as evidence that the SUT behaved correctly [61], and significant differences are evidence of the contrary.

The test outcome of a system can be unpredictable because of non-determinism, which means that such systems are instances of the oracle problem. SHT was developed to resolve this specific type of oracle problem [70]. SHT recognises that such systems may have a typical output distribution, and that information about this typical output distribution may be available prior to execution, even if information about the test outcome of a single execution is not. Since it conducts testing by checking the SUT's output distribution against the typical output distribution, it can be applied in situations where it is not possible to predict the test outcome of a single execution.

2.6.1 Assumptions

The generalisability of SHT is limited [176], because the technique makes several assumptions that may not always hold. For example, the SUT or input generation method must be non-deterministic [123]. Thus, the technique is not applicable to scenarios in which the SUT is deterministic, and random testing is not used. Another example of such an assumption is that the expected output distribution is known [122]. One could use reference implementations (RIs) to determine the expected distribution [70], if this assumption does not hold. Unfortunately, the negative issues that are associated with the use of RIs may also affect SHT, if this approach is used e.g. correlated failures. Another issue could be that an RI may not be available [70], thus the technique might not always be applicable.

The statistical techniques used in SHT make assumptions about the data. This means that some statistics may not be applicable to certain data samples that are produced by a system because these data samples may not satisfy the assumptions of these statistics. To illustrate, Ševčíková et al. [61] investigated a simulation package, and found that the data that was produced by this system either adhered to Normal or Poisson distributions. Parametric statistics assume that the distribution

is Normal, and so may not be applicable to all of the data samples produced by their simulation package.

In situations in which a test statistic's assumptions have been broken, one could use a different statistic that does not make such an assumption. For example, one could use a non-parametric statistic, if the data is abnormally distributed. However, it has been reported that non-parametric statistics are less effective [72], thus doing so may compromise the effectiveness of SHT. Alternatively, it might be possible to modify data samples to satisfy the broken assumptions. For example, Ševčíková et al. [61] used a test statistic that assumed that variance was constant across all dimensions of an output, but remarked that such an assumption may not always hold. They also stated that log or square root transformations could be used to stabilise the variance [61]. Thus, performing such transformations may resolve the issue.

2.6.2 Effectiveness

Ševčíková et al. [61] compared the performance of Pearson's χ^2 with a statistic they called $LRTS_{poisson}$, and found that the latter was more powerful. This suggests that the effectiveness of SHT is partly dependent on the choice of statistical test, and that one should always opt to use the most effective, applicable statistical tests.

The summary statistics that characterise the distributions are also an important determinant of effectiveness. To illustrate, Guderlei et al. [72] found that variance was more effective than mean. Interestingly, they also observed that the variance and mean detected mutants that the other failed to detect. This indicates that one should use multiple summary statistics.

SHT's performance was abysmal in an experiment conducted by Guderlei et al. [72]. In this experiment, SHT only considered characteristics of the SUT's output, instead of the entire output. The authors suspect that this explains SHT's performance. This suggests that one should maximise the amount of data being considered by SHT to enhance its effectiveness. However, one of the findings of an experiment conducted by Ševčíková et al. [61] was that tests that considered fewer dimensions of the output could be more effective. This indicates the converse i.e. reducing some of the data being considered by SHT could improve effectiveness. These conflicting observations suggest that the most appropriate amount of data to expose SHT to is context dependent. We believe that future work that establishes a set of guidelines with respect to the most apposite amount of data to make available to SHT would be valuable.

Yoo [202] exposed a variant of SHT, called Statistical Metamorphic Testing (see Section 2.6.3), to different datasets. He observed that the dataset that offered the worst performance may have had outliers, and suggested that this may explain its comparatively poorer performance to other datasets. This suggests that the nature of the data is also important.

SHT is necessary, but not sufficient; false positives and negatives are possible [61]. In SHT, one has control over the significance level. Higher significance levels result in more false positives, but fewer false negatives [61] and vice versa. Thus, one can tune the significance level to enable better management of these classification errors.

It is unclear which test cases are incorrect [122]; thus manual inspection of each is necessary. This suggests that reducing the size of the sample might be beneficial from a debugging perspective. However, it has also been observed that increasing the sample size can lead to an increase in the

number of faults that can be detected by the technique [72]. Thus, reducing the size of the test suite could lead to a reduction in effectiveness. Unsurprisingly, it has been reported that SHT can be very resource intensive because it requires a large number of executions to produce stable results [72]. This means reducing the test suite size might also be beneficial from an efficiency viewpoint, but doing so may compromise the stability of the technique. There are clearly several trade-offs associated with the sample size that might affect the effectiveness of the technique.

2.6.3 Statistical Metamorphic Testing

Recall that SHT assumes that one either has knowledge about the expected output distribution, or a reference implementation that can determine the expected distribution. Guderlei and Mayer [70] combined SHT with MT to ameliorate this assumption. The integrated approach is called Statistical Metamorphic Testing. The approach operates as follows. For a given MR, the source and follow up test cases are executed multiple times to obtain two or more sets of outputs. Each set is aggregated into one statistical value, and a statistical hypothesis test is evaluated based on these values. This integrated approach also enhances MT's capability to operate in non-deterministic systems [202].

The integration of these techniques can clearly be advantageous in some respects e.g. from a generalisability perspective. However, the union of these techniques can also be detrimental in other ways. For example, it was reported that in Statistical Metamorphic Testing, the most appropriate statistical analysis is dependent on the MR [202]. This means one must expend additional effort to determine the most appropriate statistical analysis for each MR, which is an otherwise unnecessary task in standard MT.

Yoo [202] investigated the effectiveness of Statistical Metamorphic Testing and found that it is affected by choice of statistical hypothesis test and choice of test cases. He also noted that Statistical Metamorphic Testing was incapable of detecting faults that failed to propagate to the output i.e. cases of coincidental correctness.

2.7 Comparing techniques

Sections 2.2 to 2.6 described a series of techniques that were devised to alleviate the oracle problem. Each technique was explored in terms of its effectiveness, efficiency, and usability. Sections 2.7.1, 2.7.2, and 2.7.3 compare these techniques on the basis of these issues.

2.7.1 Effectiveness

Certain faults can only be detected by assessing specific oracle information e.g. specific test cases may be necessary to detect certain faults. Table 2.5 shows that different techniques have access to different oracle information, and thus may find different faults. For example, since Assertions only evaluates the SUT based on oracle information from a single execution, it cannot detect faults that require oracle information from multiple executions. Statistical Metamorphic Testing has access to such information and so can detect such faults. However, some MRs place restrictions on which test cases can be used. Let TCF be the set of test cases that can manifest a particular fault, F. If an MR's restrictions prevent it from using members of TCF, then it will not be able to detect F. Assertions do not have this restriction, and so might be able to detect F. Clearly, practitioners should select testing

Technique	**Can this technique experience correlated failures?**	**Is this technique effective for coincidental correctness?**	**Examples of choices that one can make with respect to the design and application of an oracle based on this technique.**	**Examples of ways in which the coverage of this technique is restricted, in terms of oracle information.**
N-Version Testing	True	False	One can choose whether the programming language that an RI is written in, is the same as the programming language of the SUT.	N-version Testing is a black-box testing technique, and so it cannot use white-box oracle information.
Metamorphic Testing	True	False	One can choose the tightness of an MR.	Some MRs place restrictions on source test cases. This means the code coverage of such MRs might also be restricted.
Assertions	Unknown	Unknown	One can decide whether or not to introduce Auxiliary Variables.	Assertions are limited to checking a single execution. Thus, Assertions cannot draw oracle information from multiple executions, simultaneously.
Machine Learning Oracles	True	False	One has a choice over which feature extractors will be used by the technique.	This is a black-box testing technique, and so is restricted to verifying the correctness of the SUT's output.
Statistical Hypothesis Testing	True	False	One has a choice over which statistical tests will be conducted by the technique.	If the SUT is deterministic, then Statistical Hypothesis Testing mandates that one uses random testing, as the test case generation strategy.

Table 2.5: A summary of the effectiveness data for each technique, based on Sections 2.2 to 2.6.

techniques based on the types of faults that their system is prone to. This highlights some research opportunities; in particular, it may be possible to extend the types of faults that one technique can detect, by combining it with another technique that uses different oracle information. An example of this was presented at the end of Section 2.4.1.

Although different techniques can find different types of faults, they might not be able to detect all instances of these faults. Every technique has limitations in terms of coverage (see Table 2.5), thus this potential explanation applies to all of the techniques. Alternatively, correlated failures may explain this phenomenon for a subset of the techniques (see Table 2.5). A large amount of research has been conducted on reducing correlated failures for N-version Testing, but very little has been done in the context of other techniques that are known to experience correlated failures. We therefore believe that such research could be a valuable asset to the community.

Table 2.5 outlines an example of a design and application option for each technique. Sections 2.2 to 2.6 revealed that some techniques have more design and application options than others. Such techniques offer a greater degree of control; this might enable better optimisation of the technique for different contexts. However, it may be more difficult to find a suitable design and mode of application for such techniques.

One's choices regarding a technique's design and application options can have both a positive and negative impact. For example, increasing the number of feature extractors used by an ML Oracle can lead to improvements in one type of classification, but reductions in another. Unfortunately, guidelines on how one should exploit many of these types of design and application options for their context have not been proposed. Research that leads to the establishment of such guidelines would be useful.

Unfortunately, to the best of our knowledge, empirical data regarding the effectiveness of Assertions for coincidental correctness is unavailable. We therefore believe that significant value can be gained by studying this technique in the context of coincidental correctness and the oracle problem. Interestingly, Sections 2.2 to 2.6 suggest that the remaining techniques can be ineffective for coincidental correctness (see Table 2.5). Thus, research that explores methods of reducing the impact of coincidental correctness on these techniques would be valuable. For example, Clark and Hierons [48], and Androutsopoulos et al. [3] developed a series of metrics that estimate the probability of encountering coincidental correctness on particular program paths. Such metrics can be used to select test cases that are less susceptible to coincidental correctness.

2.7.2 Efficiency

Table 2.6 reveals that the contexts in which the different techniques perform particularly poorly may differ. For example, the feature extractors that are available in a certain context may be particularly inefficient, but the SUT in this context may have very few large arrays. This means that assertions and machine learning may be efficient and inefficient in this context respectively. Conversely, in another context, the SUT may contain an abundance of these programming constructs, and so assertions may be inefficient, but the feature extractors available in this context may be efficient.

Technique	Examples of reasons that may explain why the efficiency of this technique might vary in different contexts.
N-Version Testing	RIs must be developed to replicate the functionality of the SUT, but they do not necessarily have to mimic other quality attributes, including efficiency. Thus, in some situations, one might develop an RI to be equally (or more) efficient to the SUT, but in another situation, one might opt to disregard efficiency completely.
Metamorphic Testing	Different MRs have different MTG sizes. Therefore, MT's efficiency in a particular context will be partly determined by the MTG sizes of the MRs that are applied in that context.
Assertions	Assertions can be inefficient at testing large arrays. Thus, the overall efficiency of Assertions in a particular context, will be determined by the number of large arrays in the SUT that must be checked by the technique.
Machine Learning	Some feature extractors are more efficient than others; since the appropriate choice of feature extractors is domain specific, the efficiency of ML may vary in different domains.
Statistical Hypothesis Testing	There is a trade-off between efficiency and result stability (which is determined by sample size). In one situation, the tester might require greater result stability than in another, and thus, might have to sacrifice efficiency to a greater extent in such a situation.

Table 2.6: A summary of the efficiency data for each technique, based on Sections 2.2 to 2.6.

2.7.3 Usability

Table 2.7 demonstrates that the required effort to apply each technique can vary. Techniques may differ in terms of the contexts in which they are difficult to use. For instance, it may not be possible to obtain an RI via component harvesting for the SUT, and so manual construction of an RI may be necessary in a certain context. In the same context, all of the assumptions of a statistic being used in SHT may be satisfied by the data, so data transformation tasks are unnecessary. N-version Testing may require substantial effort in such a scenario, but SHT may not. The converse is also possible.

Table 2.7 also shows that the required expertise for different techniques also varies. Thus, one's choice of technique may partly depend on the expertise currently available. For example, if one lacks knowledge about machine learning, but has an adequate understanding of statistics, then one may be more inclined to select Statistical Hypothesis Testing, instead of Machine Learning oracles.

2.8 Coincidental Correctness

The Reachability, Infection, and Propagation (RIP) model [106], also referred to as the Propagation, Infection, and Execution (PIE) model [185], was formulated to describe conditions that are necessary for fault detection. In particular, the model states that fault detection is predicated on the following

Technique	Examples of reasons that may explain why the usability of this technique might vary in different contexts.	Skills and knowledge that might be required to use this technique.
N-Version Testing	In some situations, it might be necessary to implement an RI from scratch, but this may not be necessary in other situations.	The capability to write programs in different programming languages.
Metamorphic Testing	The tester may have to spend additional time and effort studying the domain to acquire domain knowledge in situations in which a domain expert is not available.	Requires domain knowledge.
Assertions	The amount of effort required to write assertions in a given context will depend on the programming language being used in that context.	Requires domain knowledge.
Machine Learning	Certain ML tasks are necessary in some situations, but are unnecessary in others e.g. labelling dataset items.	Knowledge about machine learning.
Statistical Hypothesis Testing	Tasks like data transformation may be necessary in some situations, but not others.	Knowledge about statistics.

Table 2.7: A summary of the usability data for each technique, based on Sections 2.2 to 2.6.

three conditions: (1) execution of a faulty program statement, (2) resultant infection of a program state, and (3) propagation of this state to the output. The first two conditions are necessary to manifest misbehaviour and the last condition is needed to detect it. The RIP model was later extended to include a fourth condition, which is that a test oracle must be able to reveal the failure by inspecting the output. This extended version of the RIP model is called the Reachability, Infection, Propagation, and Revealability (RIPR) model [106].

The RIP model is a useful means of describing different types of coincidental correctness. Suppose that a test case tc_s was executed, and satisfied the first two conditions, but failed to satisfy the third condition. In other words, tc_s executed the faulty program statement, which led to the infection of a program state, but this infected program state did not propagate to the output. This scenario is referred to either weak or strong coincidental correctness [119]. Let tc_w be another test case, in which the first condition has been satisfied, but the second and third condition have not. Thus, the faulty program statement has been executed by tc_w, but this did not lead to the infection of a program state. This scenario is referred to as weak coincidental correctness [119]. Weak coincidental correctness subsumes strong coincidental correctness. We use the term "coincidental correctness" to refer to weak coincidental correctness in this thesis.

Information flow strength describes the percentage of information that propagates between two program points; a higher percentage indicates greater strength. Thus, information flow strength determines the likelihood that an infectious state will propagate to the output and by inference the chance of observing coincidental correctness. Masri et al. [119] investigated weak information flow strength and found that it's very prevalent. They examined the information flow strength of six real world systems and discovered that between 63.76 - 97.58% of the information flows had a strength of 0. It's therefore not surprising that they also found that coincidental correctness is widespread [120, 119]. In particular, 96.5% and 72% of the seeded versions of 10 subject programs they investigated, in other studies, had weak and strong coincidental correctness respectively [120, 119]. A similar analysis was conducted by Xue et al. [197], who obtained similar results.

Let MUT be a faulty version of the system under test, SUT. Mutation testing is a technique that can generate MUT from SUT, by injecting an artificial fault into SUT. MUT is said to have been killed by strong mutation testing, if the outputs of MUT and SUT differ. Weak mutation testing is said to have killed MUT, if the internal program states of SUT and MUT differ. Androutsopoulos et al. [3] recognised that coincidental correctness is present in situations where a mutant has been killed by weak mutation testing, but not strong mutation testing. Such situations have been demonstrated to be prevalent [144, 143].

As discussed in Chapter 1, coincidental correctness can limit the effectiveness of testing, equivalent mutant detection and debugging techniques. Thus, the ubiquity of coincidental correctness has motivated research on limiting its impact on such techniques. The remainder of this section presents this research.

2.8.1 Test case selection

Different program paths have different information flow strength, which means that coincidental correctness is more likely to manifest on certain program paths [48]. This has motivated the development of several test case selection strategies that prioritise program paths that are less susceptible to coincidental correctness.

One such strategy was devised by Apiwattanapong et al. [4]. They developed a regression testing technique called MATRIX. MATRIX leverages Dependence Analysis and Symbolic Execution to identify code that has changed between versions and analyses these changes to deduce the necessary conditions required for the changes to propagate to the output. Test case selection is then restricted by these conditions. Such test cases cannot be susceptible to coincidental correctness.

Although MATRIX can successfully select test cases that are not affected by coincidental correctness, it cannot be applied outside of the context of regression testing. Hierons [75] devised a strategy that can be applied more generally. Boundary Value Analysis (BVA) is a test case generation strategy that partitions the input domain into a set of subdomains, $D = \{SD_1, SD_2, ..., SD_n\}$, such that the behaviour of the SUT is similar for all inputs that belong to a particular subdomain, SD_i, but different from inputs in another subdomain, SD_j. BVA selects test inputs that are at the boundaries of these subdomains. Such test inputs may be susceptible to coincidental correctness. Hierons proposed selecting two adjacent subdomains, SD_i and SD_j, and sampling test input values x and y from SD_i and SD_j respectively, such that x and y are geometrically close. The tester can execute a specified behaviour in the SUT with x and y to determine whether it behaves differently in response to these

inputs; if it does not, then x and y are susceptible to coincidental correctness, and so should not be used.

An alternative strategy, called Dynamic Impact Analysis, was developed by Goradia [66]. Let s_i and s_j be two program statements in the SUT, such that s_j is dependent on s_i. Dynamic Impact Analysis estimates the probability that a failure that propagates from s_i to s_j will cause the value of s_j to be faulty. This is repeated for all pairs of statements in the SUT that have this dependency relationship. This information can be used to determine the overall likelihood that a failure will propagate to the output (i.e. coincidental correctness) for a particular test case. Thus, Dynamic Impact Analysis can be used to prioritise test cases that are less susceptible to coincidental correctness.

Similarly, Clark and Hierons [48] devised a means of quantifying the likelihood of coincidental correctness affecting a test case, through an analysis of information flows in the system. Clark and Hierons [48] define a collision as a program point where the behaviour of the system can cause two or more different states to transition to the same state and realised that collisions were a necessary condition for coincidental correctness. They therefore developed a metric called squeeziness that estimates the chance of encountering coincidental correctness by looking at the number of collisions in the system. Similarly, Androutsopoulos et al. [3] developed five metrics that are also based on the information flows within the system. Some of these metrics were found to be highly correlated with coincidental correctness (Spearman ρ: 0.95). Such metrics can be used to direct test effort to avoid coincidental correctness.

Chen et al. [32] also developed a metric that can be used to predict coincidental correctness. Let $s = a\#b$ be a program statement, such that a and b are two program variables and # is an arbitrary arithmetic or logic operator. Chen et al. [32] recognised that the probability that the value of s is correct varies, depending on the correctness of a and b. In particular, the probability that the value of s is correct can be different in the following four scenarios: both a and b are correct ($Scenario_1$), a is correct, but b is not ($Scenario_2$), b is correct, but a is not ($Scenario_3$), and a and b are both incorrect ($Scenario_4$). Chen et al. [32] introduced a metric that capitalises on this observation to estimate the probability that coincidental correctness is present in a particular test case. Such information can be used to prioritise test cases that are less susceptible to coincidental correctness. The accuracy of this metric was hindered by certain control flow constructs in the system. This limitation was later rectified by Zhou et al. [211].

Finally, Laski et al. [104] proposed using mutation testing to introduce a corrupt state into the system, and checking if the corrupt state propagates to the output. Failure to propagate to the output indicates that the test case is susceptible to coincidental correctness, and so should not be used.

2.8.2 Mutation Testing

In the context of mutation testing, coincidental correctness can be described as follows: Let S_o be the SUT and S_m be a non-equivalent mutant. Also let s_m denote the state in S_m after the mutated statement executes and s_o be the corresponding state in S_o. Coincidental correctness occurs if s_m and s_o map to the same output, despite the differences in code.

Despite the prevalence of coincidental correctness, little research has been conducted on determining mutant equivalence in the context of coincidental correctness. To our knowledge, only one approach has been proposed. Offutt and Lee [144] propose comparing the outputs of S_o and S_m, as

well as comparing s_o with s_m. If the outputs of S_o and S_m are the same, but s_o with s_m differ, then coincidental correctness has been identified.

2.8.3 Debugging

A sizeable number of approaches have been proposed, that attempt to alleviate the impact of coincidental correctness on Spectrum-based Fault Localisation (SBFL). This section explores these approaches.

2.8.3.1 Clustering-based strategies

Clustering-based strategies involve grouping executions together based on their behavioural similarities [128]. It is assumed that executions that exercise the same fault exhibit similar behaviours, and thus will be grouped together by clustering. This enables one to identify passed executions that have similar behaviours to failed executions; such executions are considered to have a high chance of being affected by coincidental correctness [127].

Miao et al. [127] implement such an approach with K-Means Clustering. If a particular cluster c_i contains more failed than passed executions, all passed executions in c_i are deemed to be coincidentally correct. Such executions can be relabelled to failed, or removed. The extent to which SBFL can be improved by their technique varies from negligible to substantial [128].

This approach is susceptible to classification errors e.g. genuine passing executions may be grouped in clusters that are dominated by failed executions, or a cluster may be dominated by coincidentally correct test cases. Li and Liu [107, 108] developed two alternative extensions for the technique, that might alleviate this problem. The first extension combines the suspiciousness scores of executions within each cluster, to determine the overall suspiciousness of each cluster. One is restricted to only considering the most suspicious clusters. The other method involves using suspiciousness scores to estimate the number of coincidentally correct executions that are present. This estimate can be used to restrict the number of passed test cases that can be reclassified as coincidentally correct.

Masri and Assi [118] developed a suite of related techniques, some of which were based on clustering. In their first technique, Technique-I, program statements that appear in a large number of failed executions, and a small number of passed executions are considered to be likely to be correlated with coincidental correctness. Let $CCPS$ be the set of all such program statements, and $CCTC$ be the set of all passed executions that executed at least one member of $CCPS$. Technique-I classifies all test cases in $CCTC$ as coincidentally correct [118].

For each test case tc_i in $CCTC$, their second technique, Technique-II, computes the probability that tc_i is coincidentally correct, based on the number of statements in $CCPS$ that are executed by tc_i and the suspiciousness of these statements. Technique-II only classifies test cases in $CCTC$ as coincidentally correct, if they have a particularly high probability [118].

Their third technique, Technique-III, partitions $CCTC$ into two clusters, based on the similarity of the most suspicious $CCPS$. The suspiciousness of each cluster is determined by computing the average suspiciousness of all $CCPS$ in that cluster. The test cases that are in the most suspicious cluster are marked as coincidentally correct by Technique-III [118].

Masri and Assi developed a fourth technique, which they call Tech-I [119]. In Tech-I, all passing and failing test cases are partitioned into two clusters, based on the similarity of the most suspicious

$CCPS$. All passing test cases that are in the cluster with the highest proportion of failed test cases are labelled as coincidentally correct.

Finally, the last technique they proposed, Tech-II, is an extended version of Technique-III that implements fuzzy set membership. In particular, the extent to which a statement is a member of $CCPS$ is based on the proportion of passed test cases, and the proportion of failed test cases it was executed in [119]. The clustering approach was modified to account for this.

An alternative approach was proposed by Yang et al. [198]. They suggest clustering executions based on the suspiciousness scores of their statements. Passed test cases that are members of clusters that also contain failed test cases are classified as coincidentally correct.

The approaches discussed above involve applying clustering algorithms to data that have high dimensionality i.e. program spectra [62]. Weishi and Mao [188] realised that such data could have a negative impact on the efficiency of these algorithms. This motivated them to devise an approach that reduces the dimensionality of program spectra. Their approach involves grouping program statements that are commonly executed together. Such groups are referred to as Dynamic Basic Blocks. A simplified version of the program spectra is constructed for each test case execution, such that groups of statements in the program spectra are replaced by their corresponding Dynamic Basic Blocks. Clustering can then be performed on these program spectra. Again, passed test cases that are clustered with failed test cases are considered to have a high probability of being coincidentally correct.

Farjo et al. [62] recognised another limitation of applying clustering algorithms to data that has high dimensionality — the effectiveness of clustering algorithms can be adversely affected by such data. Thus, they were also motivated to reduce the dimensionality of program spectra. They proposed using Principal Component Analysis to achieve this [62].

Masri et al. [121] proposed a partially automated clustering approach, that doesn't rely on clustering algorithms. Thus, such an approach is not susceptible to the aforementioned problems relating to the dimensionality of program spectra. Their approach involves the automated generation of a Multivariate Visualisation Scatterplot. Each test case is represented by a data point on this scatterplot, and the Euclidean distance between data points communicates their similarity. The similarity between two test cases is determined by metrics that compare the test cases in terms of their execution traces. Passed test cases are represented as green data points on the scatterplot, and failed test cases are represented by red data points. The user can manually perform clustering on this graph.

2.8.3.2 Other approaches

Like many of the approaches discussed above, the approach introduced by Xue et al. [197] also leverages machine learning algorithms. In particular, their approach involves training an Ensemble-based Support Vector Machine to recognise the difference between passed and failed test cases based on their program spectra. The Ensemble-based Support Vector Machine can then be applied to the program spectra of a passed test case; the test case is deemed to be coincidentally correct, if the Ensemble-based Support Vector Machine classifies it as failed.

Another approach was devised by Bandyopadhyay and Ghosh [11]. Let p denote a passing test case. They introduced a method of estimating the probability that p is coincidentally correct. Their method involves summing the suspiciousness scores of all of the statements that were executed by p. The higher this summed value, the more likely p is coincidentally correct. Using their method, one

can obtain an estimate for each passed test case in a test suite, and consequently, can rank them based on these estimates. Let $Rank = \{p_1, p_2, ...p_n\}$ be the ranked passed test cases, such that p_1 has the highest rank and p_n has the lowest rank. They also introduce a means of estimating the number of coincidentally correct test cases that are present in the system, N, which is computed based on the number of passing test cases in which the most highly suspicious statements are present. Passed test cases p_1 to p_N in $Rank$ are marked as coincidentally correct.

One strategy for alleviating coincidental correctness in SBFL is to identify coincidentally correct test cases, and either relabel them as failed test cases, or remove them from the test suite. All of the approaches discussed above leverage this strategy. Alternatively, one can retain such test cases and establish mechanisms to reduce the impact of coincidental correctness in these test cases.

Such a strategy was adopted by Wang et al. [187]. Let f denote a particular fault type. Wang et al. [187] observed that the set of data and control flows that immediately precede and succeed the execution of a fault of type f, is typically the same as the set of data and control flows that immediately precede and succeed the execution of another fault of type f. These typical data and control flows are said to be the context pattern of f. They develop context patterns for several common fault types. Their approach involves using the context patterns to refine program spectra. In particular, statements in the program spectra that do not exhibit any of the context patterns are marked as unexecuted. This means that cases where the faulty statement was executed, but did not result in a failure (coincidental correctness) are removed from the program spectra.

Zhang et al. [208, 207] also adopt such a strategy. They model the SUT as a Control Flow Graph. Each edge in the graph is associated with a suspiciousness score, which is based on the number of passing and failing executions that the edge is exercised in. They introduce an algorithm that calculates the overall suspiciousness of each statement in the graph, based on the suspiciousness of the edges it is connected to. This suspiciousness score incorporates information about the propagation of suspicious states, and so accounts for coincidental correctness [208].

One final approach that leverages this strategy was implemented by Zheng et al. [210]. Zheng et al. [210] distinguish between three types of predicates — Neutral Predicates, Fault Leading Predicates and Fault Led Predicates. The outcome of an evaluation of a Neutral Predicate is independent of the test verdict. The outcome of an evaluation of a Fault Leading Predicate is consistent across all failed and coincidentally correct test cases, but is different from the outcome of evaluating such a predicate in passed test cases. Finally, the outcome of an evaluation of a Fault Led Predicate is consistent across all passed and coincidentally correct test cases, but is different from the outcome of evaluating such a predicate in failed test cases. They demonstrate that various analyses based on the differences of passed and failed executions can be performed to classify predicates. They also introduce a new suspiciousness score calculation that incorporates information about each predicates type, to account for coincidental correctness.

Finally, Zhang et al. [205] adopted a unique strategy. They recognised that coincidental correctness only affects passed test case executions. To that end, they developed a variant of SBFL that only leverages information from failed test cases. Such an approach cannot be affected by coincidental correctness.

2.9 Threats to validity

This section outlines the main threats to validity and how they were mitigated. Threats are organised by Mapping Study phase.

2.9.1 Search

Since the first author was unfamiliar with the problem domain at the outset, relevance misclassifications were possible. To reduce this possibility, edge case papers were conservatively kept for more detailed analysis, after more knowledge had been accrued.

Many of the titles and abstracts did not give sufficient information about the true intent or scope of the paper, which may have led to misclassifications. Authors of known relevant papers were emailed with our list of their relevant papers, and requested to confirm comprehensiveness. This reduced the impact of this threat.

Another threat is the restrictions placed on the search e.g. number of research repositories. These were necessary to retain feasibility. To reduce the impact of these restrictions, we applied several other search strategies e.g. perusing reference lists.

The search facilities offered by many repositories were flawed, which means they may not have returned all relevant studies. Where necessary, a series of workarounds were used to address this problem e.g. using Google's "site:" function for ACM DL.

There are also threats to repeatability; web content is ever growing, and thus the ranking of web pages are ever changing, which means that 50 consecutive irrelevant results may appear prematurely in comparison to the first search, or after significantly more results have been examined.

Including grey literature is an important step to combatting publication bias [99] and obtaining cutting edge research. We used research repositories like Google and Citeseerx to obtain such literature.

Determining the relevance of a paper is a subjective task. To reduce subjectivity, an inter-rater reliability test was conducted independently by two researchers on the Relevance Inclusion and Exclusion Criteria, on a sample of 12 papers. The results of this test were used to increase the precision of our criteria.

2.9.2 Data Extraction

The nature of the data being captured was broad, and none of the available data extraction forms were flexible enough to capture all of the important data. Thus, a data extraction form was specifically developed for this Mapping Study, with appropriate inbuilt flexibility.

Some of the papers did not report all of the data that was necessary to complete the data extraction form, and we were unable to elicit some of this data from the authors of these papers. In such cases, it was necessary to make assumptions about the data. Although these assumptions were informed, there is a chance that they may have been incorrect.

2.9.3 Quality Criteria

None of the available quality instruments were suitable; adoption of inappropriate quality instruments may lead to inaccurate classifications. Thus, an appropriate quality instrument was developed. The

design of our quality instrument was based on the guidelines of Kitchenham [99], and took inspiration from 27 examples of quality instruments, and domain knowledge.

Measuring the quality of a paper involves some degree of subjectivity. To that end, two researchers independently conducted a test of inter-rater reliability on the quality instrument, on a sample of 12 papers. We used the results of this test to fine tune our quality criteria.

2.9.4 Throughout the process

Many decisions were necessarily subjective; several practices were adopted to decrease potential bias introduced through subjectivity. For example, as mentioned above, inter-rater reliability tests were conducted on several critical, subjective parts of the process. The review protocol was also defined prior to starting the Mapping Study, which enabled most subjective decisions to be taken before the data had been explored.

Additionally, where possible, subjectivity in processes was reduced through careful design e.g. the relevance Inclusion and Exclusion Criteria are based on relatively objective guidelines.

We contacted the authors of the papers that were covered by the mapping study, at various stages of the process, by email, to elicit information and/or provide confirmation on various issues. We found that, in some cases, it was not possible to contact the author, and that a large proportion of the authors did not reply (an author is not considered to have replied if the author did not reply within a month of the last email that was sent). Given that there was such a large number of authors, human error is also possible i.e. we may have failed to email a small number of them. Additionally, even though many of the authors did reply, some of their responses only addressed a subset of the issues. This could affect our results e.g. people that we did not establish contact with might have had a paper that could have been included in the mapping study, or had people addressed all of the issues, making certain assumptions about their work would not have been necessary.

2.10 Conclusion

Although several Systematic Literature Reviews that target associated areas exist, each one explores the subject matter from a different perspective and thus offers a distinct contribution. For example, many systematic reviews had different scopes, which means they surveyed different sets of papers. For example, Kanewala et al. [94], Nardi and Delamaro [139], and Baresi and Young [12] had a more constrained scope; they were restricted to Scientific Software, Dynamical Systems, and specification- and model-based testing respectively. Harman et al. [74] had a wider scope e.g. they accounted for non-automated solutions like crowd sourcing. However, they had a different relevance criteria and search strategy, so their systematic review procured different studies.

Different systematic reviews also conducted different types of synthesis. Harman et al. [74], Kanewala et al. [94], and Nardi and Delamaro [139] conducted a higher level synthesis, which means their synthesis was effective for finding high level research opportunities e.g. measurements for oracles [74], but less capable of identifying lower level research opportunities like a technique's relationship with specific fault types. Baresi and Young [12] performed a low level synthesis, but the nature of their data is different e.g. they explored multiple specification languages from a high level view, instead of a finer grained inspection of issues that generalise to all specifications. Finally, some systematic reviews

have additional or different objectives. For example, Pezzè and Zhang [154] and Oliveira et al. [146] endeavoured to establish a taxonomy to classify oracles and Harman et al. [74] examined trends in research on the oracle problem.

Since our Mapping Study takes a unique perspective on the Oracle Problem in terms of the combination of scope, type of synthesis and objectives, it also offers a distinct contribution. In particular, our Mapping Study surveyed the literature on automated testing techniques that can detect functional software faults in non-testable systems. It also presented a series of discussions about each technique, from different perspectives like effectiveness and usability, performed a set of comparisons between these techniques, and identified research opportunities.

A key observation made by the Mapping Study was that most of the techniques that had been covered are ineffective for coincidental correctness. The effectiveness of the other techniques that were covered by the Mapping Study for coincidental correctness has not been empirically demonstrated. This motivated our exploration of research that had been conducted on coincidental correctness — Section 2.8 outlines this research. This exploration revealed that coincidental correctness is widespread, very little research had been conducted on testing in this context, and that all of the testing solutions that had been proposed were related to test case generation — there were no test oracle based solutions. We therefore believe that research that offers an oracle-based solution to this problem would be a substantial contribution. This motivated Objective 1 of the thesis, and thus the research that is described in Chapter 3.

The Mapping Study also found that Metamorphic Testing is the most widely studied technique for alleviating the oracle problem, and that this technique can be negatively affected by coincidental correctness. These observations, in conjunction with the research that indicated that coincidental correctness is prevalent (see Section 2.8), motivated Objective 2, and by implication, the research that is described in Chapter 4.

Our exploration of the research on coincidental correctness also demonstrated that very little research had been conducted on coincidental correctness in the context of mutation testing, despite its prevalence, and importance in this context. This partly motivated Objective 3, and the research that is described in Chapter 5. Objective 3, and the research that is described in Chapter 5 were also partly motivated by the lack of solutions for alleviating the impact of non-determinism on mutation testing — see Section 5.4. Finally, our exploration of coincidental correctness also revealed that Spectrum-based Fault Localisation techniques can be negatively affected by coincidental correctness. The ubiquity of this problem motivated our final objective, Objective 4, and the work that is described in Chapter 6.

Chapter 3

Interlocutory Testing

As discussed in Section 1.1.2, coincidental correctness can compromise the effectiveness of traditional testing techniques. Section 2.8 revealed that coincidental correctness is widespread, and that this has motivated research on mitigating coincidental correctness in testing. However, it also indicates that all of this research revolves around test case generation strategies — no oracle-based solutions have been developed, to our knowledge. In this chapter, we introduce such a solution — Interlocutory Testing; see Section 3.1. Thus, this chapter attempts to address Objective 1 (see Section 1.1). A series of experiments were conducted to determine the feasibility, effectiveness and generalisability of Interlocutory Testing. A description of these experiments can be found in Section 3.2, the results in Section 3.3, and threats to validity in Section 3.5. Most of the relevant related work for this chapter was presented in Chapter 2; this material is supplemented in Section 3.4. Finally, conclusions are drawn in Section 3.6.

In summary, the following contributions are made in this chapter:

- Interlocutory Testing: a testing technique that can operate under the influence of coincidental correctness.
- Probabilistic Interlocutory Relations: A method for conducting Interlocutory Testing on functionality that is governed by chance, when coincidental correctness is present.
- 48, 4, 1, 1 and 3 oracles based on Interlocutory Testing for the following respective programs: a Genetic Algorithm for the Bin Packing Problem, Dijkstra's Algorithm, Bubble Sort, Binary Search and Knuth-Morris-Pratt.
- An evaluation of the technique's feasibility, effectiveness and generalisability based on five case studies.
- A comparative analysis of the effectiveness and usability of Interlocutory Testing and traditional testing techniques.

3.1 Interlocutory Testing — Technique Description

Interlocutory Testing was designed to perform testing in systems that are susceptible to coincidental correctness. The technique is introduced in Section 3.1.1, and Section 3.1.2 demonstrates how it can be extended to cope with non-determinism.

3.1.1 Interlocutory Testing and Coincidental Correctness

The following running example is used throughout this section. The SUT, Sys, is a Genetic Algorithm, which is a search optimisation technique. The SUT consists of the following major components: Initial Population Generator, Crossover, Mutation, and Selection.

Algorithm 1: Selection Operator

Input: A numeric value, PS, that denotes the maximum population size, and a $Population$, such that $Population.size() \geq PS$.

Output: A modified version of $Population$

```
1 Initialisation code;
2 //Population.add(generateRandomIndividual());
3 Initialisation code;
4 while Population.size() ≠ PS do
5     SelectedIndividual = selectRandomIndividual(Population);
6     Population.remove(SelectedIndividual);
7 end
```

Let Sys_{so} denote the Selection component of Sys. Algorithm 1 is the implementation of Sys_{so}. According to Algorithm 1, the input for Sys_{so} consists of a numeric value, PS, which denotes the maximum population size, and a $Population$, such that $Population.size() \geq PS$. Let $Population_{SOI}$ denote the state of $Population$ just after Line 3 has executed. Lines 4 − 7 in Algorithm 1 outline the process used by Sys_{so} to modify $Population$. In particular, Sys_{so} iteratively removes random elements of $Population$ until $Population.size() == PS$. The output of Sys_{so} is the state of $Population$, after it has been subjected to this process; $Population_{SOO}$ denotes this state.

Let Sys^f be a faulty version of Sys. In particular, the implementation of Sys_{so} in Sys^f is a version of Algorithm 1, in which Line 2 is uncommented. The faulty line in Sys^f (Line 2 in Algorithm 1) causes a random individual to be erroneously added to $Population_{SOI}$ during the initialisation phase of Sys_{so}. Since Sys_{so} iteratively removes random individuals from $Population$ until $Population.size() == PS$, all traces of an additional member being added to $Population_{SOI}$ might be lost by the time the execution reaches the $Population_{SOO}$ state. Thus, Sys^f is susceptible to coincidental correctness.

3.1.1.1 Intuition

The relationship between an input and output can be used to predict execution trace behaviours, and discrepancies between these predictions and the execution trace can reveal coincidentally correct faults. This is the intuition and basis of Interlocutory Testing. The passages that follow will exemplify the intuition, by illustrating how it can be used to detect the fault in Sys^f.

Suppose that Sys^f was executed, and that details of the execution trace were captured in a log file, LOG. The execution trace of Sys_{so} is a subsequence of LOG. Suppose that $Population_{SOI}$ and $Population_{SOO}$ were extracted from this subsequence, and were designated the $Input$ and $Output$ respectively. One relationship that might exist between $Input$ and $Output$ is $Population_{SOI}.size() > Population_{SOO}.size()$.

Such a relationship can be used to reason about how the SUT should have behaved (i.e. how Sys would have behaved). To illustrate, in a situation in which $Population_{SOI}.size() > Population_{SOO}.size()$, one would expect the selection operator to have removed individuals from the population.

Since one would be aware that the Sys_{so} is intended to iteratively remove random individuals from $Population$ until $Population.size() == PS$, and that the Crossover Operator is intended to be the only means by which individuals can be added to a $Population$ of size PS, it follows that the Crossover Operator should have generated some individuals and added them to $Population$. In particular, the Crossover Operator should have generated the same number of individuals, as were removed by the Selection Operator (which should be $Population_{SOI}.size() - Population_{SOO}.size()$). The results of such reasoning can be used to devise predictions about aspects of LOG. In continuation of the example above, one can predict that the Crossover Operator generated $Population_{SOI}.size() - Population_{SOO}.size()$ individuals.

$CrossoverN$ denotes the number of individuals that were generated by the Crossover Operator during the execution. $CrossoverN$ is reported in LOG. The prediction above can be checked against LOG by verifying $CrossoverN == Population_{SOI}.size() - Population_{SOO}.size()$. This predicate would evaluate to false because an additional individual would have been added to $Population_{SOI}$ by Sys^f; this indicates that the prediction is incorrect, and thus that a fault is present. The prediction would have been correct if the fault had not been present.

To reiterate the intuition behind Interlocutory Testing, in the context of the running example; the relationship between an input and output ($Population_{SOI}.size() > Population_{SOO}.size()$) can be used to predict execution trace behaviours ($CrossoverN == Population_{SOI}.size() - Population_{SOO}.size()$), and that discrepancies between these predictions and the execution trace can reveal coincidentally correct faults (the fault in Sys^f).

Having introduced and demonstrated the intuition behind Interlocutory Testing, we will finally discuss why such an approach can find coincidentally correct faults. Consider the following; let ETB_f and ETB_c be two execution trace behaviours that map to a plausible output O. In a correct implementation of the system, ETB_c is used to produce O. In the context of the example above, ETB_c is the initialisation procedure of Sys_{so} in Sys (i.e. Lines 1 to 3 in Algorithm 1), and O is $Population_{SOO}$. Coincidental correctness occurs when a different behaviour (ETB_f) is used instead of ETB_c to produce O. In this case, ETB_f is the initialisation procedure Sys_{so} in Sys^f (i.e. Lines 1 to 3 in Algorithm 1). Since Interlocutory Testing checks whether ETB_c was used to produce O, it directly tests coincidental correctness.

3.1.1.2 Technique Description

Algorithm 2: Interlocutory Relation

Input: Execution trace log file LOG

Output: Pass/Fail verdict $IRVerdict$

```
1  Let IORs be a set of Input-Output Relationships;
2  Let IORVerdicts be an empty list;
3  for IOR_i ∈ IORs do
4      IORVerdict = IOR_i.assessIOR(LOG);
5      if IORVerdict = Satisfied then
6          Let IDs be the set of Interlocutory Decisions that are associated with IOR_i;
7          Let IDVerdicts be an empty list;
8          for ID_i ∈ IDs do
9              IDVerdict = ID_i.assessID(LOG);
10             IDVerdicts.add(IDVerdict);
11         end
12         if IDsAreSatisfied(IDVerdicts) then
13             IORVerdict = SUTPossiblyCorrect;
14         else
15             IORVerdict = SUTFaulty;
16         end
17     else
18         IORVerdict = Inconclusive;
19     end
20     IORVerdicts.add(IORVerdict);
21 end
22 IRVerdict = IORsAreSatisfied(IORVerdicts);
```

An oracle in Interlocutory Testing is called an Interlocutory Relation (IR). Algorithm 2 outlines the procedure for evaluating an IR, and thus how Interlocutory Testing realises the intuition described in Section 3.1.1.1. In this section, we leverage the running example described above, to explain this procedure in detail.

According to the intuition behind Interlocutory Testing, the relationship between an input and output can be used to predict execution trace behaviours. From an implementation perspective, this can be achieved by associating an input and output (Input-Output pair) with a prediction about aspects of the execution trace LOG. Devising predictions for every individual Input-Output pair would be impractical. To that end, IRs use Input-Output Relationships (IORs) to group Input-Output pairs together. Certain predictions are applicable to all Input-Output pairs in such a group. Consider the running example; $Population_{SOI}.size() > Population_{SOO}.size()$ is an IOR, and the following prediction can be made for all Input-Output pairs that are grouped by this IOR: the Crossover Operator produced $Population_{SOI}.size() - Population_{SOO}.size()$ individuals. Let IOR_1 denote this IOR.

The term "Interlocutory Decision" (ID) is used to refer to a prediction that is associated with an IOR. An ID can take any form, as long as it can unambiguously express one's prediction and be auto-

matically compared with the execution trace LOG. For example, as demonstrated in Section 3.1.1.1, predicates can be used to describe the prediction e.g. $CrossoverN == Population_{SOI}.size() - Population_{SOO}.size()$ (this ID is associated with IOR_1). Other forms of description may include Program Spectra [196], Slices [73] or UML Diagrams [103]. Different forms of description are apposite for different situations. For example, Program Spectra excel at describing control flow behaviours, but are less useful for state data. Conversely, predicates are excellent for state data but are not as proficient at describing control flow behaviours.

In Interlocutory Testing, the SUT is executed to produce an execution trace, LOG; LOG serves as input into Algorithm 2. Lines 4 – 19 in Algorithm 2 outline how a single IOR can be evaluated. In particular, on Line 4, the $Input$ and $Output$ are extracted from LOG and are used to check whether the execution satisfied the IOR. With regards to the running example, $Input = Population_{SOI}$, $Output = Population_{SOO}$, and IOR_1 is satisfied if $Input.size() > Output.size()$. Lines 5 – 16 state that if the IOR is satisfied, then the IR assesses all of the IOR's associated IDs against LOG (e.g. $CrossoverN == Population_{SOI}.size() - Population_{SOO}.size()$ would be assessed if IOR_1 was satisfied), and if the IDs are collectively satisfied, then the IOR concludes that the SUT might be correct (SUTPossiblyCorrect), or otherwise, reports that the SUT is faulty (SUTFaulty). Note that the definition of "collectively satisfied" depends on the IOR e.g. it may be necessary for all IDs to be satisfied in some IORs (as is the case in the running example), but it may be acceptable if only a subset of the IDs are satisfied in another IOR.

We call an Input-Output pair I/O valid, if a correct version of the SUT can produce output O in response to input I. IOR_1 clearly doesn't cater for all valid Input-Output pairs i.e. $Population_{SOI}.size()==Population_{SOO}.size()$ is possible in a correct version of the SUT. Under such circumstances, IOR_1 can only report that the test was inconclusive (Lines 17 – 19 in Algorithm 2 cater for this). An IR can be designed to contain multiple IORs. Thus, this can be remedied by creating more IORs that cover such pairs. For example, $Population_{SOI}.size() == Population_{SOO}.size()$ can be IOR_2 and $CrossoverExecuted == false$ can be its ID. Lines 1 – 3 and Line 20 in Algorithm 2 allow the IR to apply the process described across Lines 4 – 19 for one or more IORs (in this case IOR_1 and IOR_2), and keep a record of their SUTPossiblyCorrect/SUTFaulty/Inconclusive verdicts. The IR can then make a decision on the SUT's correctness (i.e. SUTFaulty or SUTPossiblyCorrect) based on these verdicts (Line 22). The exact implementation of this decision procedure varies for different IRs. In the case of an IR that contains both IOR_1 and IOR_2 (henceforth referred to as IR_1), if one verdict is SUTFaulty, then the final verdict of IR_1 is SUTFaulty, and if at least one verdict is SUTPossiblyCorrect, and none are SUTFaulty then, the final verdict of IR_1 is SUTPossiblyCorrect.

IRs that contain multiple IORs, can define potentially complex relationships between the IORs to enhance their effectiveness. To illustrate, since IOR_1 and IOR_2 collectively cover all valid Input-Output pairs, if a situation arises in which neither IOR_1 nor IOR_2 are satisfied (both verdicts are Inconclusive) i.e. $Population_{SOI}.size() < Population_{SOO}.size()$, then IR_1 can be certain that the Input-Output pair under consideration is not valid and can thus report SUTFaulty as its final verdict.

3.1.1.3 Implicit IDs

Interlocutory Testing has three major phases. Firstly, during an execution of the SUT, Interlocutory Testing leverages logging functions to capture and store relevant (for the evaluation of IRs) aspects of

the execution trace in a log file. Secondly, relevant data from this log file, that can be used to evaluate an IR, is extracted. To illustrate, consider the example above; the SUT was executed to produce LOG (Phase 1), and $Population_{SOI}$, $Population_{SOO}$, $CrossoverN$ and $CrossoverExecuted$ were subsequently extracted from LOG to evaluate IR_1 (Phase 2). In the final phase, the IR is evaluated based on this execution trace data. Phases 2 and 3 are repeated for each IR evaluation.

Interlocutory Testing may transform execution trace data during these phases. For example, during Phase 1, $Population_{SOI}$ might be translated into an XML format, so that it is in an appropriate form for logging. In continuation of this example, Interlocutory Testing might convert the XML data pertaining to $Population_{SOI}$ into an object, during Phase 2, and in Phase 3, the size of $Population_{SOI}$ may be computed based on this object. In an alternative example, Interlocutory Testing might compute and store the size of $Population_{SOI}$ during Phase 1. In such a situation, the aforementioned transformation tasks described for Phase 2 and Phase 3 are unnecessary during these phases.

A fault could exist in the SUT that might cause Interlocutory Testing to crash during Phase 1. For example, suppose that $Array[i] = CrossoverExecuted$, and that a fault exists that causes $Array = null$. Since $CrossoverExecuted$ is required by IR_1, the logging function will attempt to access $Array[i]$. This can lead to an array out of bounds error. Similarly, faults in the SUT may lead Interlocutory Testing to crash during Phase 2. The following example illustrates this. The crossover operator leverages an iterative process to add new individuals to the population. Let us suppose that the logging function logs $CrossoverN$ on the last iteration of this iterative process. A fault may exist that prevents the execution of the last iteration, and thus the logging function may not log $CrossoverN$. During Phase 2, Interlocutory Testing would crash when it attempts to extract $CrossoverN$ from LOG, since $CrossoverN$ would not exist in LOG. Finally, it is also possible for Interlocutory Testing to crash during Phase 3, as a result of a fault in the SUT. To illustrate, suppose that an IR iterates PS number of times over a set of array indexes, and that these indexes are used to access elements of $Population_{SOO}$. A fault in the SUT may cause $Population_{SOO}$ to contain fewer members than PS, and this would cause an array out of bounds error during the evaluation of the IR.

In essence, the reason that Interlocutory Testing can crash during these phases is because certain execution trace behaviours that were expected to manifest, failed to do so because of a fault in the SUT. Such behaviours are "implicit" Interlocutory Decisions, and a crash during these phases indicates that the execution failed to satisfy these IDs. Let IR_i be an IR. If Interlocutory Testing crashes while logging execution trace data for IR_i (Phase 1), extracting data from LOG for the evaluation of IR_i (Phase 2), or while evaluating IR_i (Phase 3), then an "implicit" ID was not satisfied, and resultantly, the final verdict of IR_i is SUTFaulty.

3.1.2 Interlocutory Testing and Non-determinism

In this section we introduce an SUT and an IR for this SUT; these will serve as the running example.

Algorithm 3: Tournament Selection Operator

Input: A numeric value, PS, that denotes the maximum population size, a $Population$, such that $Population.size() \geq PS$, and another numeric value TS that denotes the tournament size.

Output: $Winners$

1 Let $Winners$ be an empty list;
2 **while** $Winners.size() \neq PS$ **do**
3 $tournament = formRandomTournament(Population, Winners, TS)$;
4 $tournament$ consists of a set of competitors $tournament = \{Competitor_1, Competitor_2, ..., Competitor_n\}$. One $Competitor_i \in tournament$ is randomly selected to be the winner of the tournament; this competitor is denoted as $winner$. The chance of a particular $Competitor_i$ being randomly selected to be the winner is $\frac{Competitor_i.getFitnessValue()}{AggregatedFitness}$, such that $AggregatedFitness$ is the sum of fitness values that are associated with all members of $tournament$;
5 $Winners.add(winner)$;
6 **end**

The SUT for the running example is a genetic algorithm. One of the components of the SUT is the Tournament Selection Operator (TSO); Algorithm 3 describes the implementation of TSO. TSO's input consists of three variables: PS and $Population$, which were introduced earlier, and TS, which is a numeric value that corresponds to the tournament size. Lines 3 – 5 and 1 of Algorithm 3 describe the process for conducting one tournament. A tournament is a set of competitors $tournament = \{Competitor_1, Competitor_2, ..., Competitor_n\}$. The first step of the process is to randomly generate a tournament; Line 3 of Algorithm 3 achieves this. Each $Competitor_i \in tournament$ has a fitness value. Each competitor has a chance of winning the tournament, that is based on their fitness value, relative to the aggregated fitness values of all other competitors in the tournament. Thus, even though any competitor could win, the competitor with the greatest chance of winning is the one with the highest fitness value. The second step of the process involves randomly selecting one $Competitor_i \in tournament$ to be the winner of the tournament (Line 4 of Algorithm 3 caters for this); $winner$ denotes the selected competitor. The final step of the process simply consists of recording the winner of the tournament — this is achieved by Lines 1 and 5 of Algorithm 3. Because of Line 2 of Algorithm 3), TSO performs PS number of tournaments. The output of TSO is the set of individuals that were selected to be winners across these PS tournaments.

Having introduced the SUT above, we now describe an IR for this SUT, which will be henceforth referred to as "TournamentPIR". As discussed above, an invocation of TSO leads to PS number of tournaments being performed. Let $AllTournaments$ be a set containing these tournaments. $tournaments$ is a set of pairs $\langle tournament_i, winner_i \rangle$, such that $tournament_i \in AllTournaments$, and $winner_i$ is the winner of that tournament: $tournaments$=$\{\langle tournament_1, winner_1 \rangle, \langle tournament_2, winner_2 \rangle, ..., \langle tournament_{PS}, winner_{PS} \rangle\}$. TournamentPIR consists of one IOR, IOR_{TPIR}, that is only satisfied when the following condition has been met: For each $\langle tournament_i, winner_i \rangle$ in $tournaments$,

$tournament_i$ contains at least two competitors, $Competitor_j$ and $Competitor_k$, such that $Competitor_j$ $.getFitnessValue() \neq Competitor_k.getFitnessValue()$.

Let $tournaments_{strong}$ be a subset of $tournaments$, such that for each $\langle tournament_i, winner_i \rangle \in tournaments_{strong}$, $winner_i$ was the competitor with the highest fitness in $tournament_i$. Conversely, let $tournaments_{weak}$ be a subset of $tournaments$, where in each $\langle tournament_i, winner_i \rangle \in tournaments_{weak}$, $winner_i$ was the solution that had the lowest fitness. IOR_{TPIR} may be associated with an ID that predicts that $tournaments_{strong}$ contains more members than $tournaments_{weak}$.

In summary, when every tournament in $tournaments$ has at least two competitors with different fitness values (this is the IOR), TournamentPIR predicts that $tournaments_{weak}$ will contain fewer tournaments than $tournaments_{strong}$ (this is the ID). Although it is unlikely, $tournaments_{weak}$ can validly contain more tournaments than $tournaments_{strong}$, which means that TournamentPIR can incorrectly conclude SUTFaulty. We refer to this type of conclusion as a false positive.

3.1.2.1 Intuition

The example above demonstrates that IRs that deal with probabilistic behaviours require an alternative evaluation method, to reduce their susceptibility to false positives. We refer to such IRs as Probabilistic IRs (PIRs). For the sake of clarity, IRs that use the evaluation method described in Section 3.1.1 will henceforth be referred to as Deterministic IRs. In this section, we present the intuition behind the alternative evaluation method used by PIRs.

The discussion above illustrates that certain behaviours can cause the evaluation of a PIR to result in a false positive e.g. $tournaments_{strong}$ contains fewer members than $tournaments_{weak}$. The frequencies with which these behaviours occur are determined by the randomised properties of the SUT. In other words, a PIR has a typical false positive rate. The intuition behind the PIR evaluation method is to leverage statistical techniques to compare a PIR's typical false positive rate to the proportion of that PIR's verdicts that were SUTFaulty; if this proportion of verdicts is significantly higher than the typical false positive rate, then it's likely that a fault exists in the system, otherwise it is possible that the system is correct.

3.1.2.2 Technique Description

Algorithm 4: PIR Evaluation Method

Input: A test suite that contains n test cases $ts = \{tc_1, tc_2, ..., tc_n\}$, a numerical value FPR_{tc} that denotes the typical false positive rate for a test case, and another numerical value FPR_{ts} that denotes the typical false positive rate for a test suite.

Output: $Verdict$

1 Let $TCVerdicts$ be an empty list;
2 **foreach** $tc_i \in ts$ **do**
3 Let $count(SUTFaulty_{tc_i})$ and $count(SUTPossiblyCorrect_{tc_i})$ be the total number of times the IR reported SUTFaulty and SUTPossiblyCorrect in tc_i respectively;
4 $R_{tc_i} = count(SUTFaulty_{tc_i}) \div (count(SUTFaulty_{tc_i}) + count(SUTPossiblyCorrect_{tc_i}))$;
5 $StatisticalTest = Pearsons\chi^2(R_{tc_i}, FPR_{tc})$;
6 **if** $R_{tc_i} > FPR_{tc}$ *and* $StatisticalTest == Significant$ **then**
7 $PIR_C(tc_i) = SUTFaulty$;
8 $TCVerdicts.add(PIR_C(tc_i))$;
9 **else**
10 $PIR_C(tc_i) = SUTPossiblyCorrect$;
11 $TCVerdicts.add(PIR_C(tc_i))$;
12 **end**
13 **end**
14 Let $count(SUTFaulty_{TCVerdicts})$ and $count(SUTPossiblyCorrect_{TCVerdicts})$ be the total number of SUTFaulty and SUTPossiblyCorrect verdicts in $TCVerdicts$ respectively;
15 $R_{TCVerdicts} = count(SUTFaulty_{TCVerdicts}) \div (count(SUTFaulty_{TCVerdicts}) + count(SUTPossiblyCorrect_{TCVerdicts}))$;
16 $StatisticalTest = Pearsons\chi^2(R_{TCVerdicts}, FPR_{ts})$;
17 **if** $R_{TCVerdicts} > FPR_{ts}$ *and* $StatisticalTest == Significant$ **then**
18 $Verdict = SUTFaulty$;
19 **else**
20 $Verdict = SUTPossiblyCorrect$;
21 **end**

Algorithm 4 details the evaluation method that is used by a PIR (e.g. TournamentPIR) to reduce the impact of false positives. The remainder of this section explains this evaluation method.

One part of the input for Algorithm 4 is a test suite $ts = \{tc_1, tc_2, ..., tc_n\}$. Let PIR denote the PIR that is being evaluated with the evaluation method detailed in Algorithm 4. Suppose that the typical false positive rate of PIR is 30%, denoted by FPR_{tc}. FPR_{tc} is another part of the input for Algorithm 4. FPR_{tc} can be extrapolated from empirical data, be based on the tester's expertise, or be obtained from an analysis of the randomised properties of the SUT.

Lines 3 – 12 of Algorithm 4 outline a procedure for reducing the impact of false positives for a single test case $tc_i \in ts$. PIR may be evaluated multiple times during an execution of tc_i. For example, TournamentPIR is evaluated each time TSO is executed, which can happen multiple times, depending on the Generation Number parameter of the Genetic Algorithm. Each evaluation of PIR will either yield a SUTPossiblyCorrect or SUTFaulty verdict. Lines 3 and 4 of Algorithm 4 computes R_{tc_i} to be

the proportion of verdicts of PIR that are SUTFaulty in tc_i. Lines 5 and 6 of Algorithm 4 perform a comparison between R_{tc_i} and FPR_{tc} using Pearson's χ^2. Algorithm 4 then leverages Lines 6 – 12 (with the exception of Lines 8 and 11) to set $PIR_C(tc_i)$ to either SUTFaulty or SUTPossiblyCorrect, based on the outcome of this comparison. In particular $PIR_C(tc_i)$ is set to SUTFaulty if $R_{tc_i} > FPR_{tc}$ and the difference is statistically significant, otherwise, the $PIR_C(tc_i)$ is set to SUTPossiblyCorrect. To illustrate, suppose that PIR was evaluated 100 times and the result was $R_{tc_i} = 70\%$. Since $70\% > 30\%$ and the difference between R_{tc_i} and FPR_{tc} is statistically significant, $PIR_C(tc_i) = SUTFaulty$. Conversely, had $R_{tc_i} = 33\%$, the difference between R_{tc_i} and FPR_{tc} would not have been statistically significant, and thus $PIR_C(tc_i) = SUTPossiblyCorrect$. $PIR_C(tc_i)$ is effectively PIR's verdict for tc_i.

To reiterate, Lines 3 – 12 of the PIR evaluation method reduces the impact of false positives for a single test case. However, this doesn't completely mitigate the problem; $PIR_C(tc_i)$ can be a false positive due to non-determinism. We will now explain how Algorithm 4 caters for this. Recall that one part of the input for Algorithm 4 was a test suite $ts = \{tc_1, tc_2, ..., tc_n\}$. Line 2 of Algorithm 4 applies the procedure that was described across Lines 3 – 12 to each $tc_i \in ts$, and Lines 1, 8, and 11 record each tc_i's corresponding $PIR_C(tc_i)$ in $TCVerdicts$ i.e. $TCVerdicts = \{PIR_C(tc_1), PIR_C(tc_2), ..., PIR_C(tc_n)\}$. One part of the input for Algorithm 4 is FPR_{ts}, which is the typical false positive rate for $TCVerdicts$, for the PIR under consideration. FPR_{ts} can be obtained using the same methods that can be used to obtain FPR_{tc}. Lines 14 and 15 compute the proportion of SUTFaulty verdicts in $TCVerdicts$ (this proportion is denoted by $R_{TCVerdicts}$), and Lines 16 – 17 compare $R_{TCVerdicts}$ to FPR_{ts}, using Pearson's χ^2. Finally, Lines 17 to 21 set $Verdict$ to either SUTFaulty or SUTPossiblyCorrect, based on the outcome of this comparison. In particular $Verdict$ is set to SUTFaulty if $R_{TCVerdicts} > FPR_{ts}$ and the difference is statistically significant, otherwise, the $Verdict$ is set to SUTPossiblyCorrect. $Verdict$ is the final verdict of PIR.

3.1.3 Multiple IRs

We envision that, in practice, one would leverage multiple IRs. Different IRs may report different verdicts i.e. SUTFaulty or SUTPossiblyCorrect. This should be interpreted as follows: If at least one IR reports SUTFaulty, then the SUT should be considered to be faulty. The SUT should only be assumed to be correct if all IRs report SUTPossiblyCorrect.

3.2 Experimental Design

This chapter addresses the following research questions:

RQ1 Is Interlocutory Testing feasible[1]? To answer this, we investigate whether Interlocutory Testing can find faults in four real world systems.

RQ2 How effective is Interlocutory Testing? The primary objective of Interlocutory Testing is to enable effective testing in the presence of coincidental correctness.

[1] In the context of this research question, feasibility refers to whether the technique is capable of carrying out its designated task.

RQ3 Do the net gains obtained from probabilistic IRs outweigh the potential net losses? Unlike deterministic IRs, probabilistic IRs can produce false positives. It's therefore important to investigate whether the effectiveness gains offered by probabilistic IRs are not offset by the introduction of false positives.

RQ4 Which IRs should be prioritised? IRs have different characteristics. We investigate the impact these characteristics have on effectiveness. The results of this investigation should deliver insights into IR design.

RQ5 How consistent is the effectiveness of Interlocutory Testing across different test suites? The effectiveness of the technique will be partly determined by the test suite. An evaluation of the level of consistency of Interlocutory Testing's effectiveness across different test suites will provide an indication into how much of the technique's effectiveness is determined by the test suite.

We conducted three separate experiments to answer the research questions above. One experiment was designed to address RQ1. This experiment leveraged four subject programs, 40 mutants (10 mutants per subject program), 400 test cases (100 test cases per subject program), and 9 IRs (in total, across the subject programs). For ease of reference, we call this the Feasibility Experiment. RQ2 to RQ4 were handled by the second experiment, in which one larger subject program, 100 mutants, 100 test cases and 48 IRs were used. We refer to this as the Main Experiment. The third experiment was designed to address RQ5, and uses the same subject program as the Main Experiment, 29 of the mutants that were used in the Main Experiment, as well as an additional mutant, 30 test suites (one of which was the same as the one that was used in the Main Experiment), and 47 of the IRs that were used in the Main Experiment. We call this experiment the Test Suite Experiment.

The remainder of this section describes how these experiments were conducted.

3.2.1 Main Experiment

3.2.1.1 Subject Program

The subject program is a Java implementation of a Genetic Algorithm for solving the Bin Packing Problem. The subject program was developed by the author using the JAGA Genetic Algorithm API toolbox [153] and was based on a design by Mladen Jankovic [84]. We made a series of changes to the subject program to make it suitable for our experiments. In order to exercise our technique, acquisition of execution trace data is necessary. To obtain such data, we had to instrument the subject program with a logging function. We also had to make some minor modifications to the subject program's source code to accommodate the instrumentation of the logging function.

We applied Interlocutory Testing to the subject program, and found that it was capable of detecting 12 real faults. Some of these faults were inserted by the JAGA Developers, were caused by errors in Jankovics' design, the author's own errors, or a combination. A brief description of these faults can be found in Appendix B. We successfully removed 11 of these faults — our motivation for doing so can be found in Section 3.5.1. One of the real faults used double to represent precise floating point numbers which caused inaccuracies. We managed to alleviate the fault by using an alternative floating point representation, BigDecimal, but this strategy was incapable of completely eliminating the fault. Further repairs for this fault were unfortunately infeasible due to time constraints.

Some of the subject program's Java files contain a mixture of the subject program's source code and the logging function's source code. As such, it is not possible to accurately measure the size of the subject program based on the size of these Java files. Let Sys_{instru} be the subject program. To that end, we created a copy of Sys_{instru} called GAUninstru (denoted by $Sys_{uninstru}$), and removed the logging function's source code from $Sys_{uninstru}$. We measured the size of the subject program based on the size of $Sys_{uninstru}$; it consists of 1596 Source Lines of Code (SLOC)[2], 29 classes and 244 methods (average 8 per class).

Despite the fact that we leveraged a systematic approach for the identification and deletion of the logging function's source code from $Sys_{uninstru}$, we cannot guarantee that our coverage over the logging function's source code was exhaustive. Thus, $Sys_{uninstru}$ may contain a small amount of the logging function's source code. Therefore, these measurements should be interpreted as estimates. The BinPackingCrossover class in $Sys_{uninstru}$ is the largest class (consisting of 243 SLOC) that was once instrumented with the logging function's source code. We re-examined the class after a sizeable amount of time had passed and only found 4 lines of code that were a part of the logging function's source code. This suggests that very few of the logging function's lines of code were included in $Sys_{uninstru}$ and that the impact of their inclusion had a negligible impact on our estimates.

Three factors motivated the selection of this SUT. Firstly, systems development was conducted and influenced by multiple people, most being unaware of this research. This improves the representativeness of the subject program and decreases experimental bias.

Information flow strength describes the percentage of information that propagates between two program points; a higher percentage indicates greater strength. Masri and Assi [119] observed that programs with weak information flow strength are vulnerable to coincidental correctness because there's a greater chance that a corrupt program state will not propagate to the output. Since Interlocutory Testing is intended to be a general purpose testing technique that accounts for coincidental correctness, a subject with weak information flow strength is ideal because it's susceptible to standard and coincidentally correct faults. The subject has weak information flow strength because it continuously overwrites information throughout the execution.

The term "non-testable system" describes systems that have characteristics that render output prediction or comparison infeasible (see Chapter 2). Thus, testers must often weaken their expectations of the output. To illustrate, a Genetic Algorithm can produce multiple valid outputs for the same test input, so testers cannot predict the precise output. However, they may be aware of certain characteristics that a correct output has. For example, the final output only contains one solution. They may therefore deem all outputs that have this characteristic to be plausible. Clearly, the number of plausible outputs in non-testable systems can be very high. Coincidental correctness is exacerbated under these conditions because there is a greater likelihood of an infected state mapping to a plausible output. Thus, one is more likely to observe coincidentally correct faults in such systems. Our subject program is an instance of a non-testable system, and we selected it for this reason.

[2]SLOC was computed using the "Code Lines" metric in the Understand program [167]. This metric ignores blank and comment lines.

3.2.1.2 Faults

Mutation Testing tools make minor modifications to the source code [142] of the SUT to simulate real faults [2]; these augmented programs are referred to as mutants. A mutant is said to have been "killed" by a testing technique, if the testing technique detects it, or is otherwise said to have "survived". The effectiveness of a testing technique can be estimated by generating a set of mutants, M, applying the testing technique to each mutant ($m_i \in M$) and determining the proportion of mutants that were killed i.e. $\frac{KilledMutants}{TotalMutants}$, where $KilledMutants$ is the number of mutants that were killed by the technique and $TotalMutants$ denotes the total number of mutants. $\frac{KilledMutants}{TotalMutants}$ is called the Mutation Score (MS). This chapter leverages this approach, because mutation testing can ensure the availability of an adequately large sample of representative test subjects to draw meaningful conclusions from.

The MuJava mutation testing tool was used to generate a sample of random mutants [115] because it's automated and can therefore reduce experimental bias. MuJava was applied to all classes that substantially contributed to the SUT's core functionality. In particular, we excluded the test case input class, an unused class, 11 interface classes, and a class that added a minor extension to java.util.random [148] that was intended to make random number generation more convenient. We also didn't include 2 abstract classes and 3 simple data classes that stored a single object and largely implemented getter/setter methods and/or mostly exposed methods that this object already has. For example, the simple data class may have an ArrayList $ArrayObj$ and a method $remove(i)$, which simply calls $ArrayObj.remove(i)$. A comparator class was also excluded. Let S be the SUT, and M_1 and M_2 be two mutants, such that $S \neq M_1$, $S \neq M_2$, and $M_1 \equiv M_2$ [152]. Finally, Let M_3 be another mutant such that $M_3 \neq M_1$ and $M_3 \neq M_2$. Suppose that the mutant sample already contains M_1; the opportunity cost of including M_2, might be the exclusion of M_3. Thus, including multiple mutants that are equivalent to each other can reduce the diversity of faults in the mutant sample. Alternatively, suppose that M_1, M_2, and M_3 were all included in the mutant sample. Since the mutation score is calculated based on the number of mutants killed, the fault represented by M_1 and M_2 will unjustifiably contribute more to the mutation score than the fault represented by M_3. We suspect that a large number of the mutants that could be generated for the abstract, simple, and comparator classes could be equivalent to mutants that could be generated in a class that interacts with these classes. Thus, the rationale behind excluding these classes was to reduce the incidence of mutants that are equivalent to each other, and by implication the problems described above.

Unfortunately, MuJava can generate equivalent mutants; these are augmentations of the code that are equivalent to the original e.g. $x < 5$ may be modified to $x \leq 4$. The inclusion of such mutants will negatively skew the results. We manually inspected every mutant in the mutant sample to identify equivalent mutants, and subsequently removed them. Additionally, MuJava can also produce mutants that cause the SUT to crash or result in infinite loops; including these mutants would positively skew the results since the technique isn't required to detect these mutants. These mutants were also excluded. Recall that the subject program contains one real fault. This real fault could be a confounding factor for our experiment, since Interlocutory Testing may misconstrue misbehaviour emanating from it, as having originated from a mutant. One of the steps taken to remove the impact of this confounding factor included rejecting mutations to faulty code. 100 mutants were generated, because a mutant sample of this size is large enough to derive meaningful conclusions from.

Finally, we classified each mutant as coincidentally correct or non-coincidentally correct. This was

achieved as follows. An oracle was devised to test all of the output properties of the SUT. Each mutant was executed with the test suite outlined in Section 3.2.1.3 and evaluated with this oracle. A mutant is classified as coincidentally correct if this oracle does not detect the fault, because this means that the failure did not propagate to the output. Mutants that were successfully detected on the other hand, are labelled as non-coincidentally correct. The list below details the conditions that were checked by our oracle. 62 of the mutants were coincidentally correct and 38 were standard (i.e. not coincidentally correct).

- Let *OutputPop* be the output population. $OutputPop.size() = PS$.
- Let *Solutions* denote a set that contains all members of *OutputPop*, and the best individual. Also let O denote a member of *Solutions*. Finally, let *DataSet* be the set of items to be sorted into bins. O should be a permutation of *DataSet*.
- O should contain at least one bin.
- O should not contain empty bins.
- O should not contain a bin that has more items than its capacity.
- O should not have a fitness that is greater than the maximum obtainable fitness (Fitness Function Constant).

Recall that our subject program is a Genetic Algorithm, and as such, one of its components is a "Mutation Operator". Since this may be easily confused with Mutation Testing, we refer to the "Mutation Operator" component as "MO" for the sake of clarity.

3.2.1.3 Test Cases

A test case for the subject program consisted of values for 15 variables: Maximum Bin Size, Initial Number of Bins, Maximum Item Size, Number of Items, Population Size, Generations, Tournament Size, Chance of Winning Tournament, Crossover Probability, Mutation Probability, Only Accept Mutation If Better (True/False), Fitness Function Constant, ReplaceXNumberOfItems, MutationDestroy, and DataSet (see [84] for a clarification of unfamiliar variables). The following procedure was used to generate a test case: Java's standard random number generator [148] was used to generate a random value for each of the first 11 of these variables. DataSet, which is a set of items of varying sizes, was randomly generated based on these variables. Fitness Function Constant, ReplaceXNumberOfItems, and MutationDestroy were always set to 2, 3, and 2 respectively. Randomisation was used to reduce experimental bias. These random values were constrained by the following main restrictions. Upper-bounds:

- Maximum Bin Size = 28.
- Initial Number of Bins = 48.
- Maximum Item Size = 18.
- Number of Items = 48.
- Population Size = 18.

- Generations = 8.
- Tournament Size = 18.
- Chance of Winning Tournament = 100%.
- Crossover Probability = 100%.
- Mutation Probability = 100%.

Restrictions were necessary to counteract the production of infeasible test cases e.g. 1241421515346 generations. Restrictions on parameters were determined by sensitivity analysis; this involved trial and error tuning of the parameters to ensure that interesting behaviours and trends were still observable e.g. 8 generations was sufficient to observe convergence. This procedure was used to randomly generate a test suite of 100 test cases. A total of 100 test cases was deemed to be sufficient, because a test suite of this size is large enough to draw meaningful conclusions.

3.2.1.4 Measures

The Mutation Score (MS) and Failure Detection Rate (FDR) were used to measure the technique's effectiveness [193]. The former, $MS = \frac{KilledMutants}{TotalMutants}$, indicates the breadth of faults found. The latter, $FDR = \frac{FailedTestCases}{TotalTestCases}$, measures the proportion of test cases that killed a particular mutant. This is useful for identifying the likelihood of a fault being detected by the technique. These measures were used because they are accurate measures of effectiveness and are widely used by the testing community [169].

3.2.1.5 Interlocutory Relations

48 IRs were designed. The strength of some of the analyses, that were performed to answer some of the research questions (e.g. RQ4) above, is sensitive to the number of IRs that are used in this experiment. We felt that 48 IRs were sufficient to cater for such analyses. As mentioned above, a real fault was present in the system. This real fault could confound the results if any of the IRs can detect it. Some of our IRs were capable of detecting this fault. One of the steps taken to counteract this confounding factor included modifying these IRs to remove their sensitivity to the real fault. Please see Appendix A for a comprehensive list of our IRs, as well as a summary of the main aspects of these IRs.

3.2.2 Test Suite Experiment

3.2.2.1 Subject Program

One method of establishing the extent to which the effectiveness of Interlocutory Testing is consistent across different test suites includes conducting comparative analyses between different applications of the technique, such that these applications vary in terms of test suites, but have minimal variance with respect to other factors like the subject program, IRs and mutant samples. We decided to implement this method. We realised that including the test suite from the Main Experiment, henceforth referred to as TS1, in such analyses would enable us to directly draw conclusions about the generalisability of the results of the Main Experiment. We therefore decided to include TS1 in this experiment. This decision necessitates the use of the subject program described in Section 3.2.1.1 in this experiment.

3.2.2.2 Faults

Let $100NonEquiv = \{m_1, m_2, ..., m_{100}\}$ denote the 100 non-equivalent mutants that were used in the Main Experiment, and $30NonEquiv$ be a subset of $100NonEquiv$, such that $30NonEquiv$ consists of 30 mutants, and that for each $m_i \in 30NonEquiv$, $1 \leq i \leq 30$. We had intended to use $30NonEquiv$ in this experiment. However, our meta-data pertaining to what two of these mutants were had become corrupt; although this does not threaten the validity of the findings of the Main Experiment, it presents a barrier for the use of these two mutants in this experiment. We therefore refined $30NonEquiv$, by replacing these two mutants. One of the mutants was replaced with $m_{31} \in 100NonEquiv$, and the other was replaced with a similar mutant. This updated version of $30NonEquiv$ was leveraged in this experiment. Since 29 of the mutants in $30NonEquiv$ also appeared in $100NonEquiv$, we used the same experimental data that was used in the Main Experiment, to represent TS1's results for these mutants. New experimental data was obtained for TS1's remaining mutant, and all of the other test suites. Our rationale for using these mutants is that their inclusion in this experiment could increase the strength of the claims that we could make about the generalisability of the results of the Main Experiment.

3.2.2.3 Test Cases

We used a total of 30 test suites in this experiment, because this sample size is large enough to draw meaningful conclusions. As discussed in Section 3.2.2.1, one of these test suites, TS1, was the same test suite that was used in the Main Experiment. The remaining 29 test suites, referred to as TS2 to TS30, were generated using the same methodology that was outlined in Section 3.2.1.3.

3.2.2.4 Measures

We used the same measures as were outlined in Section 3.2.1.4, for the same reasons discussed in Section 3.2.1.4.

3.2.2.5 Interlocutory Relations

We had initially planned to use all 48 of the IRs that were outlined in Appendix A in this experiment. However, we discovered that one of these IRs, DecidingWhoShouldMutate, could report failures in response to the real floating point fault when exercised by one of the test suites, TS18. As discussed in Section 3.2.1.2, such reports could confound the results. An investigation revealed that this IR could only detect the real floating point fault under one very specific set of conditions, and that these conditions could only be manifested by test cases that set the mutation rate to 1. Only one such test case exists across all of the test suites — TS18. This means that the only experimental data that could have been compromised by this IR, was produced by TS18. We also found that DecidingWhoShouldMutate did not kill any mutants when it was exercised by the other test suites, and only killed one mutant when exercised by TS18. We therefore decided to omit this IR from this experiment, since doing so would mitigate the confounding factor, and would not have a material impact on the results.

3.2.3 Feasibility Experiment

3.2.3.1 Subject programs

This experiment leverages four well-known and widely used subject programs: Dijkstra's Algorithm [163], Bubble Sort [85], Binary Search [186] and Knuth-Morris-Pratt [17]. These subject programs were selected for two reasons. Firstly, since they are open source, they were developed by people that were unaware of this research; this reduces experimental bias. Secondly, each program targets a different problem, which improves the generalisability of our findings. Details about these subject programs can be found below.

Dijkstra's Algorithm

For our experiment, we used a Java implementation of Dijkstra's Algorithm that had been developed by Rosetta Code [163]. We adapted the subject program by instrumenting it with a logging function and modified the subject program's source code to accommodate the instrumentation of the logging function. We used a mutation testing tool in our experiments (see Section 3.2.3.2). Such tools can produce equivalent mutants. One obvious source of equivalent mutants includes mutations to redundant code e.g. unused methods. Another way in which we changed the subject program included removing some redundant lines of code, to reduce the incidence of such equivalent mutants.

Additionally, some of the source code was not compatible with MuJava and thus had to be modified. Several examples of such modifications include: implicit Java Generics had to be made explicit e.g. transforming ArrayList<> to ArrayList<Integer>, Ternary Operators had to be transformed into if statements and For-Each loops had to be augmented into For Loops e.g. For(Obj o: Objects) to For(int i = 0; i < Objects.size(); i++){Obj o = Objects.get(i);}.

The presence of real faults in the SUT could be a confounding factor for the experiment because the technique may mistake a failure originating from a real fault to have been produced by a mutant. The technique was therefore applied to the SUT before mutation analysis was conducted to determine whether any real faults were present. Interlocutory Testing detected a real fault in the system. The developers used a class called Vertex to represent one node in the graph. Amongst other things, a Vertex object stores data regarding the current known best distance between the node it represents and the start node. It also includes a method $compare(Vertex)$ that compares it to another Vertex object based on this distance value. During the initialisation of the algorithm, one Vertex object is created to represent the start node and is assigned a distance of 0. Additionally, one Vertex object is created for every other node in the graph, each of which is assigned a distance of infinity. These distances are updated during the algorithms execution.

A data structure called NavigableSet is used to store all of these Vertex objects. The $compare(Vertex)$ method was intended to be used by the NavigableSet collection to sort the Vertex objects based on their distances such that the Vertex with the least distance from the start node is positioned at the head of the list. The $compare(Vertex)$ method fulfils this objective. However, it also means the NavigableSet interprets two Vertex objects to be equal if they have the same distance, and since it inherits the characteristics of a Set, an object that is equal to another is deemed to be a duplicate and therefore removed. This fault led to spurious deletions of Vertex objects from the NavigableSet e.g. all but one Vertex with an infinity distance and the Vertex representing the start node were deleted

during initialisation. We rectified the fault by amending the data structures representation.

To estimate the size of the subject program, we created a copy of the subject program called DAUninstru (represented by the symbol: $Sys_{Uninstru}$), and removed the logging function's lines of code from $Sys_{Uninstru}$. We also reinstated the redundant lines of code that were removed from the subject program into $Sys_{Uninstru}$. We finally calculated the size the subject program based on $Sys_{Uninstru}$; the subject program has 7 classes, 12 methods and 168 SLOC. There is potential for this procedure to be uncomprehensive in its coverage over redundant lines of code, as well as the logging function's source code. However, as discussed in Section 3.2.1.1, our systematic approach for the identification and removal of the logging function's lines of code from $Sys_{Uninstru}$ is only likely to have missed a negligible number of lines of code and thus not have had a meaningful impact on our estimates. Only a small proportion of the subject program's source code was deemed to be redundant; thus the potential impact of failing to reinstate some redundant lines of code is likely to be minuscule.

Bubble Sort

The implementation of Bubble Sort used in our experiment was written in Java by Java2Novice [85]. We adapted the source code of this subject program by instrumenting it with the logging function. Further modifications were also made to the subject program's source code to support the instrumentation of the logging function. Finally, some redundant lines of code were also removed, to reduce the incidence of equivalent mutants.

Interlocutory Testing was applied to the subject program to check the code for real faults that could confound the results. A real fault was found; the outer for loop of the Bubble Sort algorithm executed one too many times. The impact of the fault was a reduction in performance, since the final iteration was not necessary.

We used the same procedure that was used to create DAUninstru on the Bubble Sort subject program, to obtain a copy of Bubble Sort called BSUninstru, in which the logging function's source code had been removed, and redundant lines had been reinstated. We then estimated the size of the Bubble Sort subject program based on BSUninstru. The subject program consists of 1 class, 4 methods and 36 SLOC.

Binary Search

A Java implementation of the Binary Search algorithm was borrowed from Vogella [186]. We instrumented this subject program with a logging function. Again, we used our IRs to test the program for real faults that might confound the results, and found one. Let $StartIndex_i$ and $EndIndex_i$ denote the start and end of the list on iteration i respectively. The developers determined the middle, $Middle_i$, of the list by computing $(StartIndex_i + EndIndex_i) \div 2$. $StartIndex_i$, $EndIndex_i$ and $Middle_i$ are all encoded as integers. Unfortunately, this methodology is susceptible to floating point inaccuracies and rounding errors, which means that the list can be partitioned incorrectly. We therefore implemented a more precise mechanism for partitioning the list that was not susceptible to such errors and replaced this aspect of the subject program with our new implementation.

We leveraged the approach that was used to create GAUninstru (see Section 3.2.1.1) on the Binary Search subject program, to create a copy of the subject program called BinSeaUninstru, in which the logging function's source code had been removed. We estimated the size of the Binary Search subject

program based on BinSeaUninstru. The subject program consists of 1 class, 10 methods and 135 SLOC.

Knuth-Morris-Pratt

We obtained a Java implementation of the Knuth-Morris-Pratt algorithm from Sanfoundry [17]. This subject program was modified as follows. We instrumented it with a logging function, made changes to its source code to support the instrumentation of the logging function, and removed some redundant lines of code. We leveraged the same procedure that was used to obtain DAUninstru on the Knuth-Morris-Pratt subject program to obtain a copy of the Knuth-Morris-Pratt subject program, referred to as KMPUninstru, in which the logging function's source code had been removed, and redundant lines had been reintroduced. We estimated that the Knuth-Morris-Pratt subject program consists of 1 class, 4 methods and 60 SLOC, based on KMPUninstru.

3.2.3.2 Faults

We generated 40 mutants across the four subject programs detailed above — 10 mutants were generated per program. The same mutant generation strategy that was detailed in Section 3.2.1.2, was also used in this experiment. In particular, MuJava was used to generate random mutants, equivalent and crashed mutants were rejected, and in the case of Dijkstra's Algorithm, 2 input classes were excluded from mutation testing. We used this mutant generation strategy for the same reasons that were outlined in Section 3.2.1.2.

Unfortunately, our modified version of the Binary Search subject program was incompatible with MuJava. However, the original version produced by Vogella [186] was compatible. We therefore used MuJava to generate mutants for the original subject program and translated them into the amended version. Since some of the implementation details between these versions were substantially different, these translations were necessarily approximations.

3.2.3.3 Test Cases

We generated 100 random test cases for each subject program, using the same procedure as outlined in Section 3.2.1.3, for the same reasons that were described in Section 3.2.1.3. The remainder of this section presents the main subject program specific restrictions that were used to counteract the production of infeasible test cases in this experiment.

Dijkstra's Algorithm

Dijkstra's Algorithm used the following restrictions: Maximum Edge Weight = 25, and Maximum Number of Nodes in the Graph = 100.

Bubble Sort

The following restrictions were used by Bubble Sort: List Size Lower-bound = 2, List Size Upper-bound = 99, and Maximum Element Size = 999.

Binary Search

We used the following restrictions for Binary Search: List Size Lower-bound = 1, List Size Upper-bound = 999, and Maximum Element Size = 499.

Knuth-Morris-Pratt

Let $Text$ be a string and $Pattern$ be the substring we are searching for in $Text$. Both $Text$ and $Pattern$ were restricted to characters from the alphabet (duplicates were allowed). Let $SubPattern$ denote a string, such that $SubPattern.size() \geq 2$ and $SubPattern.size() \leq 20$. To generate a test case, we first create a random $SubPattern$. We then concatenate a random number of copies of $SubPattern$ together. The upper-bound for the maximum number of copies that can be concatenated is 5. Let $PartialPattern$ denote this concatenated set of copies. Let $PatternNoise$ be a randomly generated string, such that $PatternNoise.size() \geq 2$ and $PatternNoise.size() \leq 10$. $PatternNoise$ is inserted into random parts of $PartialPattern$ a random number of times. The upper-bound on the number of insertions is 4. The resulting string is $Pattern$.

A similar procedure is used to obtain $Text$. The procedures can be differentiated as follows. $Pattern$ is used in the place of $SubPattern$ on this occasion; we refer to the resultant concatenations of copies of $Pattern$ as $PartialText$. The upper-bound on the number of concatenations on this occasion is 20. Let $TextNoise$ be the equivalent of $PatternNoise$ for $Text$. The maximum number of $TextNoise$ insertions into $PartialText$ is bounded by 29.

3.2.3.4 Measures

The measures that were detailed in Section 3.2.1.4, are used in this experiment, for the same reasons that were expressed in Section 3.2.1.4.

3.2.3.5 IRs

4, 1, 1, and 3 IRs were also generated for Dijkstra's Algorithm, Bubble Sort, Binary Search, and Knuth-Morris-Pratt respectively (see Appendices C to F for a list of these IRs, in addition to a summary of the main aspects of these IRs.). A sample of 9 IRs was deemed to be sufficiently large to support the analyses that were conducted based on this sample.

3.3 Results and Discussion

3.3.1 RQ1. Is Interlocutory Testing feasible?

This section uses the Dijkstra's Algorithm, Bubble Sort, Binary Search and Knuth-Morris-Pratt subject programs, 40 mutants, and 400 test cases to investigate the feasibility of Interlocutory Testing.

Interlocutory Testing successfully killed 39/40 (9/10 Dijkstra's Algorithm, 10/10 Bubble Sort, 10/10 Binary Search, 10/10 Knuth-Morris-Pratt) mutants and thereby obtained an MS of 97.5%. As discussed above, the technique also found real faults in 75% of the subject programs. In addition to this, false positives were not reported for any of these subject programs. This demonstrates that Interlocutory Testing is a feasible testing technique, and can be effective for different subject programs. Additionally, all of the real faults were coincidentally correct, which indicates that Interlocutory Testing can operate in the presence of coincidental correctness.

Our results also indicate that the effectiveness of Interlocutory Testing can vary for different subject programs i.e. the technique obtained an MS of 90% for the Dijkstra's Algorithm subject program, compared to 100%, 100%, and 100% for the Bubble Sort, Binary Search and Knuth-Morris-Pratt subject programs respectively. We conducted a series of Fisher's Exact Tests[3] to compare the accuracy (as measured by the number of killed and survived classifications) of the technique on every pairwise combination of subject programs. None of these tests yielded a statistically significant result ($p > 0.05$). This suggests that our results may be generalisable, since there was no significant variation across the subject programs.

We also explored the average FDR of each subject program, based on the mutants that were detected. The Dijkstra's Algorithm, Bubble Sort, Binary Search and Knuth-Morris-Pratt subject programs obtained an average FDR of 88.89%, 95.9%, 86% and 76.3% respectively. The programs clearly vary more in terms of FDR than MS. This means that, although the diversity of faults that can be found with Interlocutory Testing is similar across subjects, the likelihood of finding these faults can vary. Interestingly, however, the worst case was 76.3%, which is a very high chance. This is promising because it indicates that regardless of which subject program Interlocutory Testing was applied to, it had a high chance of detecting faults. This again, demonstrates the effectiveness of the technique and indicates that our results may be generalisable.

We would finally like to highlight that the high mutation scores that were obtained for the four subject programs studied in this section were derived from a small number of IRs — ranging from 1 to 4. This suggests that Interlocutory Testing can operate effectively with very few IRs.

3.3.2 RQ2. How effective is Interlocutory Testing?

This section, Section 3.3.3, and Section 3.3.4 leverage the Genetic Algorithm subject program, 100 mutants, and 100 test cases to answer RQ2, RQ3, and RQ4 respectively.

3.3.2.1 Real Faults

Recall that Interlocutory Testing successfully detected 12 real faults in the subject program. Details about these faults can be found in Appendix B. Some of these faults were coincidentally correct and others were not. This demonstrates that Interlocutory Testing can be effective, and can operate in the presence and absence of coincidental correctness.

3.3.2.2 Mutation Analysis

Mutation Score

Interlocutory Testing killed 87/100 mutants (49/62 coincidentally correct and 38/38 standard), and thus achieved a mutation score (MS) of 87%. The technique obtained an MS of 79% for coincidentally correct faults, which indicates that it can operate effectively in the presence of coincidental correctness. Thus, the technique has achieved its primary goal. Since these 49 faults were coincidentally correct,

[3]Fisher's Exact Test is a test statistic that can be used to compare one proportion against another. The Chi Square is another test statistic that can also perform such comparisons [151]. One key difference between these test statistics is that the former is exact, whilst the latter is an approximation [49]. The implication of this is that the Fisher's Exact Test is more accurate for small sample sizes [49], and is comparable for large sample sizes [49].

they were not detected by the standard oracle. This illustrates that Interlocutory Testing can offer substantial support for coincidentally correct faults, beyond that offered by the standard oracle.

Interlocutory Testing also obtained an MS of 100% for standard faults; this demonstrates that the technique can be at least as effective as traditional test oracles for these faults, and can thus operate effectively in the absence of coincidental correctness. The difference in Interlocutory Testing's effectiveness (as measured by the number of killed and survived classifications) for standard faults is significantly different than for coincidentally correct faults (Fisher's Exact Test: $p < 0.05$), which suggests that the effectiveness of the technique can vary for different types of faults.

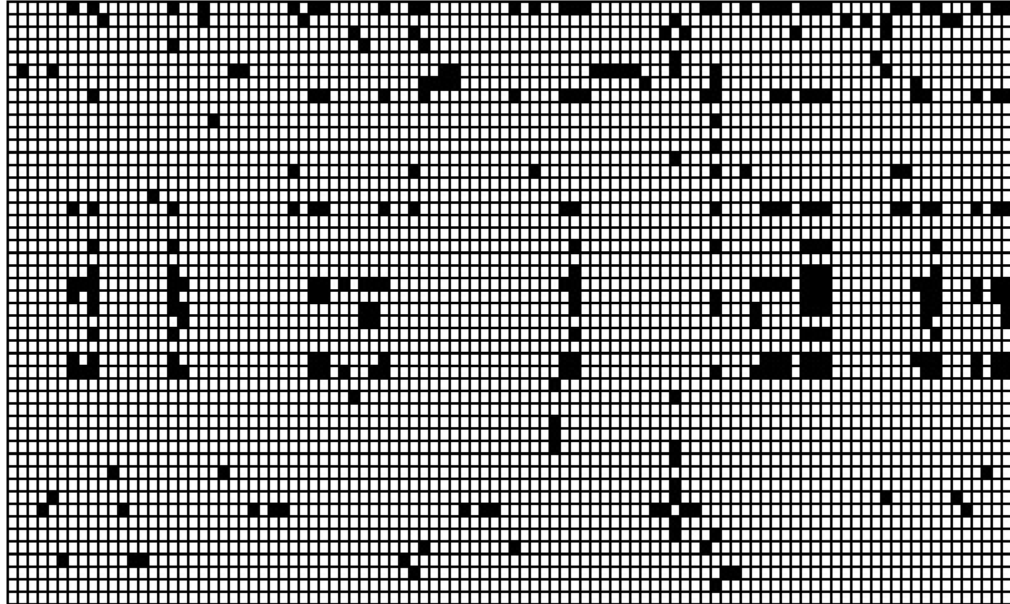

Figure 3.1: Mutants and IRs Heat Map

Figure 3.1 shows a heat map. The Y axis corresponds to IRs and the X axis represents mutants. An intersection between the X and Y axis communicates whether the relation represented by the Y axis killed the mutant on the X axis. A black square means the relation killed the mutant and a white square means the converse.

Figure 3.1 demonstrates that different IRs found a different number of faults; thus, some were more effective than others. It also shows that relations that found fewer faults could detect faults that were not detected by the more effective relations. This suggests that less effective relations may add value because they might find unique faults.

Interestingly, Figure 3.1 also shows that a large number of IRs find the same faults. This could mean that some are completely subsumed by others. To verify this, we performed a subsumption analysis to determine the minimum number of IRs required to obtain the maximum MS. We found that only 14 IRs were necessary: AverageFitnessGeneration, AverageTounamentPositionWinner, CheckIfCanReplace, ChoosingCouples, CreateRandomIndividualNewBins, CreateRandomIndividualOverflow, CrossoverRate, DecidingWhoShouldMutateFineGrained, DeduceLostItems, FFDIntegrity, MutateAllController, ShouldUseNewIndividual, TerminateGA, and TournamentComposition. This is promising because it demonstrates that Interlocutory Testing can operate effectively with a relatively small number of IRs. This supports our observations in Section 3.3.1. We believe that the development effort for 14 IRs would be acceptable in most cases.

Failure Detection Rate

The Failure Detection Rate (FDR) measure is not applicable to probabilistic IRs because these IRs base a single verdict on all of the test cases in the test suite. Thus our analysis of FDR is restricted to deterministic IRs. We additionally restricted our analysis to mutants that were detected by deterministic IRs. On average, Interlocutory Testing obtained an FDR of 64.14% (Coincidentally correct test cases: 3014/4100 and Standard test cases: 1540/3000). This demonstrates that if Interlocutory Testing could detect a fault, it was very likely to do so. Interestingly, the FDRs for standard and coincidentally correct faults were 51.33% and 73.51% respectively, which means this Interlocutory Testing was more likely to find coincidentally correct faults than standard faults. This shows that the FDR can vary for different fault types.

False Positive Rate

Recall that the verdicts of deterministic IRs are interpreted differently to the verdicts of probabilistic IRs. Since deterministic IRs shouldn't produce false positives, one can have 100% confidence in the verdict of a deterministic IR if it reports a failure. Conversely, since probabilistic IRs can report false positives, there is a chance that their failure verdicts are spurious. Section 3.1.2 described the mechanism used to curtail the false positives produced by probabilistic IRs. To briefly recap, the total number of failures reported by a probabilistic IR within and across multiple test cases are compared to typical false positive rates. If the total number of failures significantly exceeds the typical false positive rate, then the mutant is said to have been killed.

In order to test the false positive rate of our IRs, we executed a correct version of the SUT with our test suite 30 times. Thus any reported failures can be interpreted as false positives. As expected, our deterministic IRs did not report any false positives. Encouragingly, the Probabilistic IRs only reported 1/30 false positives. This shows that our mechanism for curtailing false positives was effective.

3.3.3 RQ3. Do the net gains obtained from probabilistic IRs outweigh the potential net losses?

As discussed in Section 3.3.2.2, unlike the 42 deterministic IRs, our 6 probabilistic IRs can produce false positives. In this section we explore whether the additional fault detection effectiveness offered by these Probabilistic IRs offsets their cost in terms of false positives.

The Deterministic and Probabilistic IRs detected 71 (41 coincidentally correct and 30 standard) and 56 (19 coincidentally correct and 37 standard) mutants respectively. A Fisher's Exact Test revealed that the difference in performance (as measured by the number of killed and survived classifications) between Deterministic and Probabilistic IRs was statistically significant ($p < 0.05$). This suggests that Deterministic IRs are more effective than Probabilistic IRs. Interestingly, more coincidentally correct mutants were killed by deterministic IRs than probabilistic IRs, and the converse was true for standard mutants. This indicates that each IR type was more effective than the other for different types of faults. This suggests that Probabilistic IRs can add value.

Figure 3.2 illustrates the total number of faults that were detected by both Deterministic and Probabilistic IRs and shows which of these faults were uniquely detected by the respective IR types. The graph shows that although there is a large degree of overlap (i.e. 40 faults were found by both sets of IRs), each IR type also finds a large number of distinct faults — deterministic IRs find 31

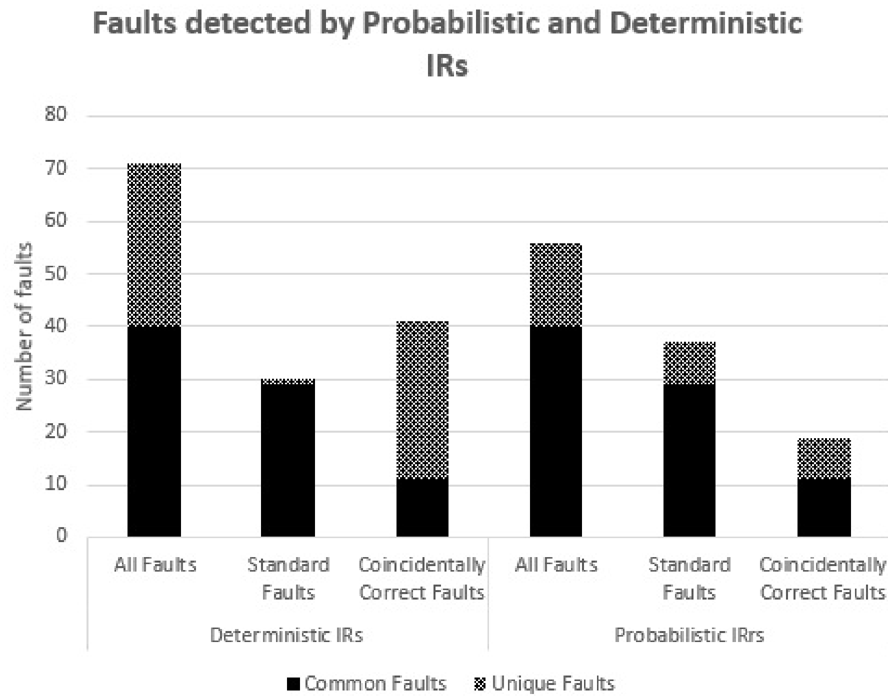

Figure 3.2: Number of faults detected by deterministic and probabilistic IRs

unique faults and probabilistic IRs find 16 unique faults. These observations provide further evidence for our hypothesis that Probabilistic IRs can add substantial value.

As discussed above, only 1/30 false positives were reported. This is negligible; given our observations above, this suggests that the benefits of using Probabilistic IRs outweigh the costs. An analysis of this false positive revealed that the AverageTounamentPositionWinner IR was responsible. This IR only detected one unique fault; thus, one could reduce the false positive rate to 0, at a cost of 1% to the true positive rate.

3.3.4 RQ4. Which IRs should be prioritised?

Different IRs have different characteristics e.g. they emphasise testing different areas of the system, use different fault detection strategies and vary in terms of the context specificness of the behaviours they predict. This section investigates the impact that these characteristics have on the technique's effectiveness and makes recommendations on which IRs should be prioritised based on the results of this investigation.

3.3.4.1 Area of the system

Although an IR can test multiple components of the SUT simultaneously, each IR tends to place greater emphasis on one particular component. We partitioned our IRs into groups, based on which component they place the most emphasis on. Figure 3.3 presents these groups. “Crossover”, “Crossover and MO”, “Fitness Function”, “GA Controller”, “Initial Population”, “MO” and “Selection” consists of 16, 10, 2, 3, 5, 8 and 4 IRs respectively. Each group is associated with a bar that communicates the total number of faults that were detected by the group. Before continuing, we would like to clarify the following; the group on the bar chart that's labelled “Crossover and MO” represents IRs that emphasise testing code that was reused across the “Crossover” and “MO” operators.

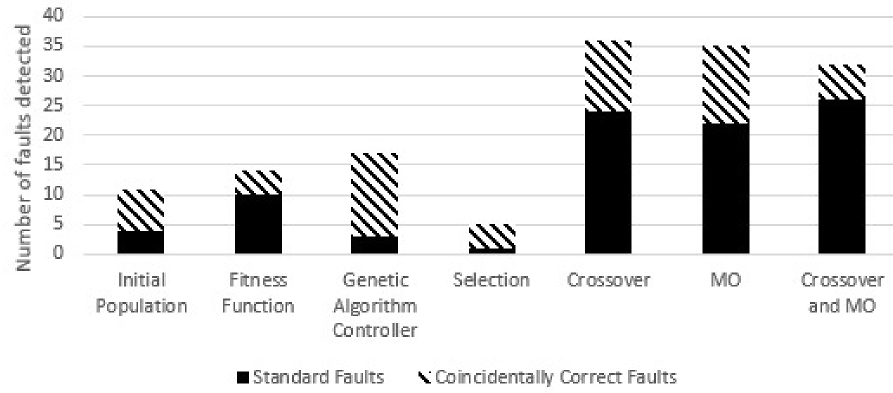

Figure 3.3: Number of mutants detected by IRs (grouped by Components)

Figure 3.3 indicates that IRs that place more emphasis on testing certain components are more effective than IRs that place more emphasis on testing other components. Interestingly, "Crossover", "Crossover and MO", and "MO" were the three most effective groups of IRs. Coupling describes the extent to which software components are interdependent. The components targeted by these three groups are highly coupled together, which suggests that IRs that target areas of the system with high coupling could be particularly effective and should be prioritised.

Interestingly, Figure 3.3 also demonstrates that IRs that emphasise testing different components find different types of faults. For example, most of the faults found by "Selection" are coincidentally correct, whilst the majority of faults found by "Crossover and MO" are standard. This suggests that the location an IR emphasises testing can also affect the types of faults that can be detected.

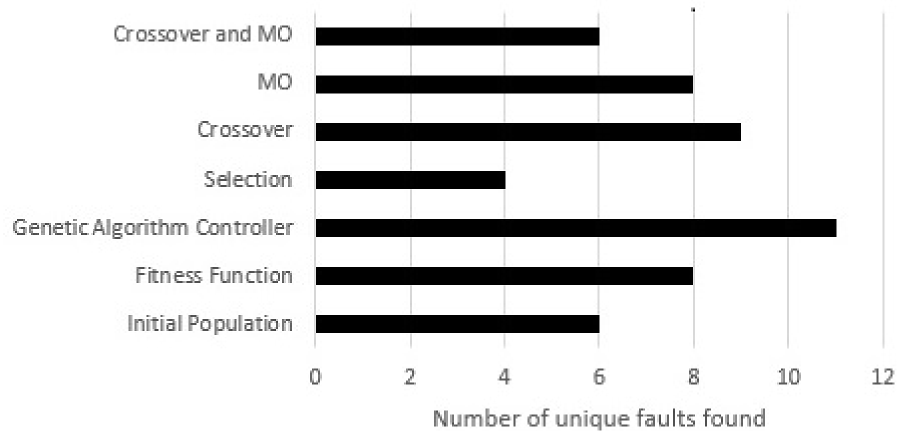

Figure 3.4: Number of unique mutants detected by IRs (grouped by Components)

According to Figure 3.4, each group of IRs found unique faults. This is not surprising because most of these unique faults were coincidentally correct (41 were coincidentally correct and 11 were standard in total), which means that failures produced by such faults don't always propagate very far. Thus, there is less opportunity for IRs that emphasise testing other components to find such faults. Therefore, value can be gained by developing IRs for components that IRs are less effective for.

3.3.4.2 IR fault detection strategies

Recall that an IR is associated with a set of IORs, IOR, and each $ior_i \in IOR$ is associated with a set of IDs, ID_i. Let IOP be an input-output pair, extracted from an execution trace, $trace_i$. Also recall

that the first step in evaluating IR involves checking whether each $ior_i \in IOR$ is satisfied by IOP. If IOP does not satisfy any IORs in IR, and IOR has full coverage over the set of all valid input-output pairs, then a failure is reported. For ease of reference we call this fault detection scenario IOR-Only-Detection. If an $ior_i \in IOR$ is satisfied, then ior_i makes predictions about the execution trace (these predictions are in the form of ID_i). These predictions are finally compared to $trace_i$; any discrepancy leads to the reporting of a failure. This fault detection scenario is referred to as IOR-ID-Detection. Also recall that an IR can report a failure, if a crash is experienced during the logging of data required for IR, extraction of execution trace data from a log file for IR, or during the evaluation of IR (see Section 3.1.1.3). Such fault detection scenarios are also classified as IOR-ID-Detection.

Let *IOROnly* be a group of IRs, such that each IR in *IOROnly* detects more mutants using the IOR-Only-Detection strategy than the IOR-ID-Detection strategy, and let $IORID$ be the converse. *IOROnly* consists of 13 IRs, and $IORID$ is composed of 18 IRs. *IOROnly* detected 33 (10 coincidentally correct and 23 standard) faults, whereas $IORID$ found 65 (37 coincidentally correct and 28 standard) faults. Thus, IRs that mainly detected faults with the IOR-ID-Detection strategy detected more faults than IRs that mostly relied on the IOR-Only-Detection strategy. We compared the effectiveness (as measured by the number of killed and survived classifications) of the IOR-ID-Detection strategy against the IOR-Only-Detection strategy, using Fisher's Exact Test, and found that the difference was statistically significant ($p < 0.05$). This indicates that IRs that mainly use the IOR-ID-Detection strategy might be more effective and thus one may consider prioritising such IRs.

A comparison of the number of killed and survived classifications made by *IOROnly* against the number of such classifications made by $IORID$ for coincidentally correct and standard faults revealed that the difference was not significant for standard faults (Fisher's Exact Test: $p > 0.05$), but was for coincidentally correct faults (Fisher's Exact Test: $p < 0.05$). This highlights the value of the IOR-ID-Detection strategy for coincidentally correct faults. Finally, we observed that both of these strategies found unique faults. In particular, *IOROnly* detects 6 faults that are not detected by $IORID$, and $IORID$ can find 38 faults that are not reported by *IOROnly*. This demonstrates that using multiple IRs that emphasise different strategies can add value.

Let PIR be a probabilistic IR. Recall that PIR may execute multiple times and thus generate a series of verdicts $Verdicts = \{v_1, v_2, ..., v_n\}$. Also recall that the final verdict of PIR is based on an evaluation of all of these $Verdicts$. A verdict $v_i \in Verdicts$ may have been determined by a different strategy than another verdict $v_j \in Verdicts$. Thus, a mixture of both strategies may be responsible for a single probabilistic IR's final verdict. This means the analysis above was not appropriate for probabilistic IRs and they were therefore omitted.

3.3.4.3 Context specificness of predicted behaviours

Recall that an IR can have multiple IORs and that these are used to make predictions about execution trace behaviours. An IR can be designed (based on the assumption that the SUT has been correctly implemented), such that when it makes predictions, it either uses all (IOR-TypeAll), or some subset of its IORs (IOR-TypeSome) to simultaneously make multiple predictions for a single evaluation. IRs were grouped according to these types; 25 and 23 IRs were classified as IOR-TypeAll and IOR-TypeSome respectively.

IOR-TypeSome IRs include multiple IORs, such that some of these IORs are satisfied in mutually

exclusive contexts. This is why such IRs cannot use all of their IORs simultaneously to make multiple predictions for a single evaluation. Recall that in order for an IOR to predict the manifestation of an execution trace behaviour, this execution trace behaviour must be expected to manifest in the contexts in which the IOR has been satisfied. Since the aforementioned IORs are not satisfied in all contexts, they can, and typically do, include predictions on the manifestation of execution trace behaviours that are not expected to manifest in all contexts. Such predictions are more context specific. By contrast IOR-TypeAll IRs either only have one IOR, or have multiple IORs that all predict behaviours in the same context. IORs in such IRs are typically designed to be applicable to most, if not all contexts and thus predict more common/general execution trace behaviours.

IOR-TypeAll and IOR-TypeSome IRs killed 49 and 81 mutants respectively. This suggests that IOR-TypeSome IRs are more effective and should be prioritised. As discussed above, since IOR-TypeSome IRs predict more context specific behaviours than IOR-TypeAll IRs, this also indicates that IRs that predict more context specific behaviours may be more effective. Although IOR-TypeAll IRs were less effective, they successfully found 6 faults that were not detected by IOR-TypeSome IRs. This suggests that it's important to use a mixture of IRs that range in terms of the execution trace behaviours they predict i.e. context specific and general behaviours.

3.3.5 RQ5. How consistent is the effectiveness of Interlocutory Testing across different test suites?

To answer this research question, we conducted a series of comparative analyses between different applications of Interlocutory Testing, such that each application of the technique varied in terms of the test suite, but not in terms of other experimental parameters like the subject program, mutants, and IRs. These comparative analyses are based on the Genetic Algorithm subject program, 30 non-equivalent mutants, 30 test suites (each consisting of 100 test cases), and 47 IRs. Further details can be found in Section 3.2.2.

Sections 3.3.5.1 to 3.3.5.3 present our comparative analyses. In particular Section 3.3.5.1 compares the test suites in terms of the mutation score, Section 3.3.5.2 performs comparisons based on the false positive rate, and FDR is the basis of the comparisons in Section 3.3.5.3.

3.3.5.1 Mutation Score

Figure 3.5 is a bar chart that shows the total number of mutants that were killed by TS2 to TS30. The X-Axis is partitioned into three subgroups that correspond to three different groups of IRs — Deterministic IRs, Probabilistic IRs, and Both IR Types. Each test suite is represented by three bars, each of which resides in a different subgroup. A bar in a given subgroup communicates the total number of mutants that were killed by the IRs that are represented by that subgroup, when exercised with the test suite that is represented by that bar. Each bar also illustrates the proportion of the faults that were killed, that were standard and coincidentally correct.

Figure 3.5 clearly shows that the Deterministic IRs obtained exactly the same mutation scores across TS2 to TS30. This suggests that one's choice of test suite has very little impact on the effectiveness of Deterministic IRs. Conversely, the table demonstrates that the effectiveness of the Probabilistic IRs did vary across test suites, and that this is more pronounced for coincidentally correct than standard faults. Interestingly however, the level of variance was relatively low, and as

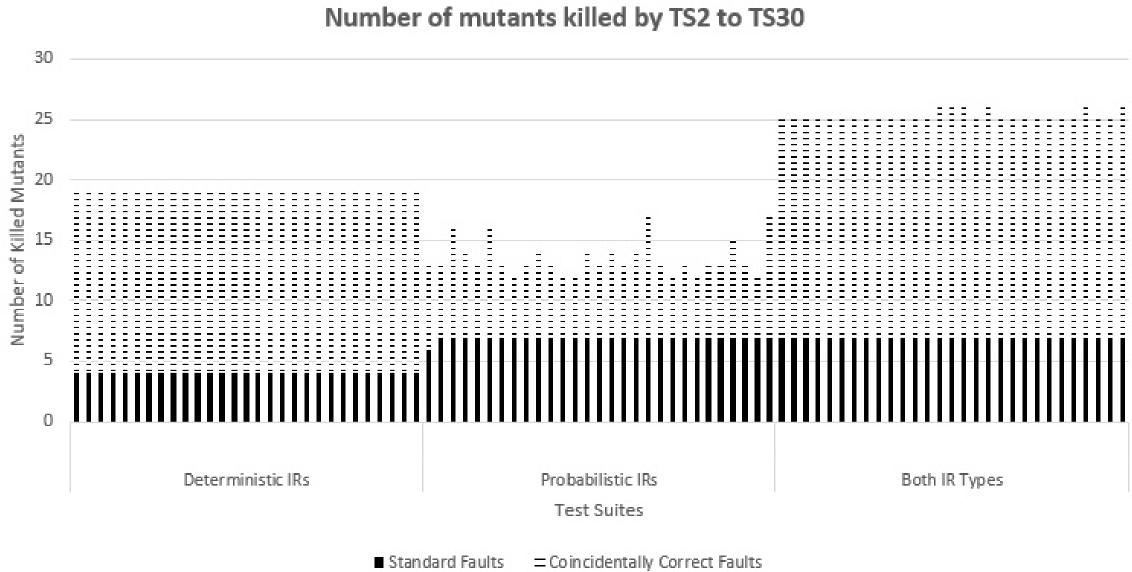

Figure 3.5: Number of mutants killed by TS2 to TS30; results filtered by IR type.

evidenced by the bars in the Both IR Types subgroup, had a very negligible impact on the overall effectiveness of Interlocutory Testing. These results indicate that the effectiveness of Interlocutory Testing is very consistent across test suites.

Table 3.1 provides another lens (in the form of descriptive statistics) on the data depicted in Figure 3.5, as well as data pertaining to the mutation scores of TS1. The table demonstrates that there was very little difference between the average effectiveness of TS2 to TS30, and the effectiveness of TS1. We conducted nine Fisher's Exact Tests, to compare the effectiveness of TS1's (in terms of the number of killed and survived mutants) Deterministic, Probabilistic, and Both IRs, against the effectiveness of TS2 to TS30's (based on the average number of killed and survived mutants across these test suites) corresponding IRs, for standard, coincidentally correct, and both fault types. We also applied the Benjamini-Hochberg correction[4] to these statistical tests, and found that none of the differences were statistically significant. This reinforces our previous observation, and suggests that the results we obtained across Sections 3.3.2 to 3.3.4.3 can generalise to other test suites.

Figure 3.6 is a bubble chart that illustrates for each IR, the extent to which this IR's effectiveness differed when it was exercised by TS1 in comparison to when it was exercised by TS2 to TS30. Each interval of the X-Axis corresponds to an IR, and the Y-Axis expresses the difference between TS1 and the other test suites, in terms of the number of killed mutants. The size of a bubble communicates the number of test suites from TS2 to TS30 that differ from TS1 to the same extent. For example, consider the bubble that is associated with the last interval of the X-Axis and -1 on the Y-Axis in Figure 3.6. The size and coordinates of this bubble can be interpreted as follows: 13 test suites from TS2 to TS30 killed one fewer mutants than TS1 with the IR represented by the last interval of the X-Axis. Finally, the graph also partitions the results by IR type.

Figure 3.6 demonstrates that, with the exception of three Deterministic IRs, all other Deterministic IRs obtained a consistent level of effectiveness across TS1 to TS30. One can observe that these three exceptional IRs vary in terms of the number of test suites that deviate from TS1 and from each other. This suggests that the likelihood of one Deterministic IR producing different results for different test

[4]In situations in which one leverages multiple statistical tests, some of the statistical tests may spuriously report a significant outcome. The Benjamini-Hochberg correction can be used to alleviate this problem [184].

Descriptive Statistics (TS2 to TS30)	Deterministic IRs			Probabilistic IRs			Both IR Types		
	SF	**CCF**	**BF**	**SF**	**CCF**	**BF**	**SF**	**CCF**	**BF**
Mean	4.00	15.00	19.00	6.97	6.55	13.52	7.00	18.21	25.21
Standard Deviation	0.00	0.00	0.00	0.19	1.43	1.43	0.00	0.41	0.41
Kurtosis	N/A	N/A	N/A	29.00	0.84	0.97	N/A	0.35	0.35
Skewness	N/A	N/A	N/A	-5.39	1.19	1.26	N/A	1.53	1.53
Minimum	4.00	15.00	19.00	6.00	5.00	12.00	7.00	18.00	25.00
Maximum	4.00	15.00	19.00	7.00	10.00	17.00	7.00	19.00	26.00
TS1's Results	*4.00*	*15.00*	*19.00*	*7.00*	*5.00*	*12.00*	*7.00*	*18.00*	*25.00*

Table 3.1: Descriptive statistics of the mutation scores obtained by TS2 to TS30, and a summary of the TS1's mutation scores. Standard Faults, Coincidentally Correct Faults, and Both Faults are represented by SF, CCF, and BF respectively.

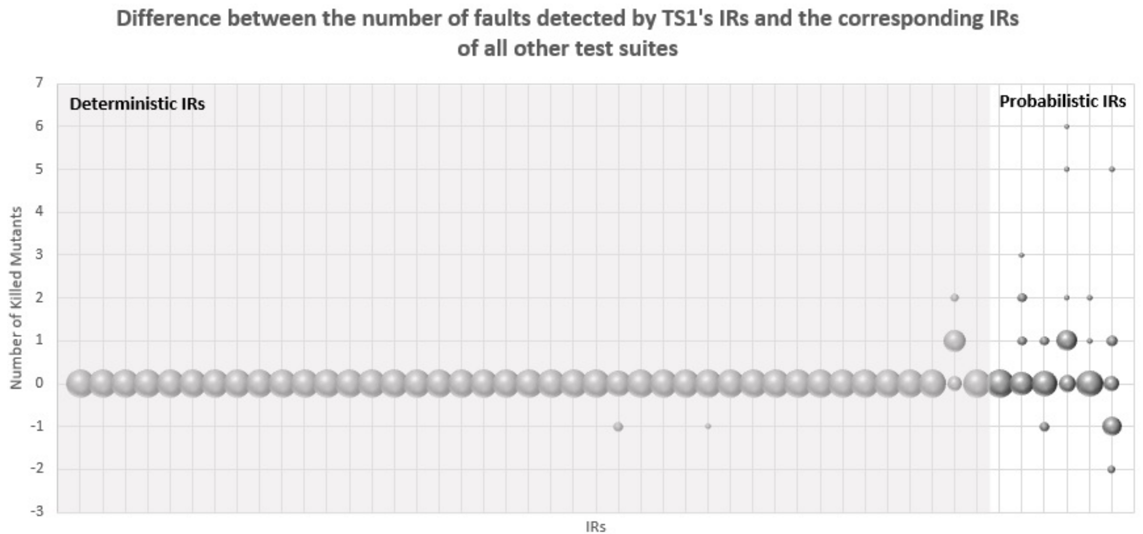

Figure 3.6: Difference in performance of TS1's IRs, against the corresponding IRs in TS2 to TS30.

suites, can be higher than for another Deterministic IR. The graph also makes apparent that, across these three exceptional IRs, the largest deviation between a test suite and TS1 was only by two faults. This indicates that, in situations in which a Deterministic IR's performance can vary across test suites, the severity of the variation is likely to be small.

As discussed above, the overall effectiveness ascertained by Deterministic IRs was consistent across all of the test suites; this means that the variations in the effectiveness of the three exceptional IRs had no impact on the overall effectiveness of our Deterministic IRs. Let IR_a be a Deterministic IR. Suppose that IR_a can detect mutant m with a test suite ts_x, but cannot detect m with another test suite ts_y. Also suppose that another Deterministic IR, IR_b, can detect m with ts_y. This demonstrates that the collective fault detection effectiveness of IR_a and IR_b remain the same for m, despite the fact that the effectiveness of IR_a can vary for m. This example illustrates why the overall effectiveness of our Deterministic IRs remained constant across test suites despite the minor variations in the

effectiveness of several individual IRs. The example also highlights the value of developing multiple IRs that overlap in terms of the faults that they can detect i.e. had IR_b not been present, then m might not have been detected by ts_y.

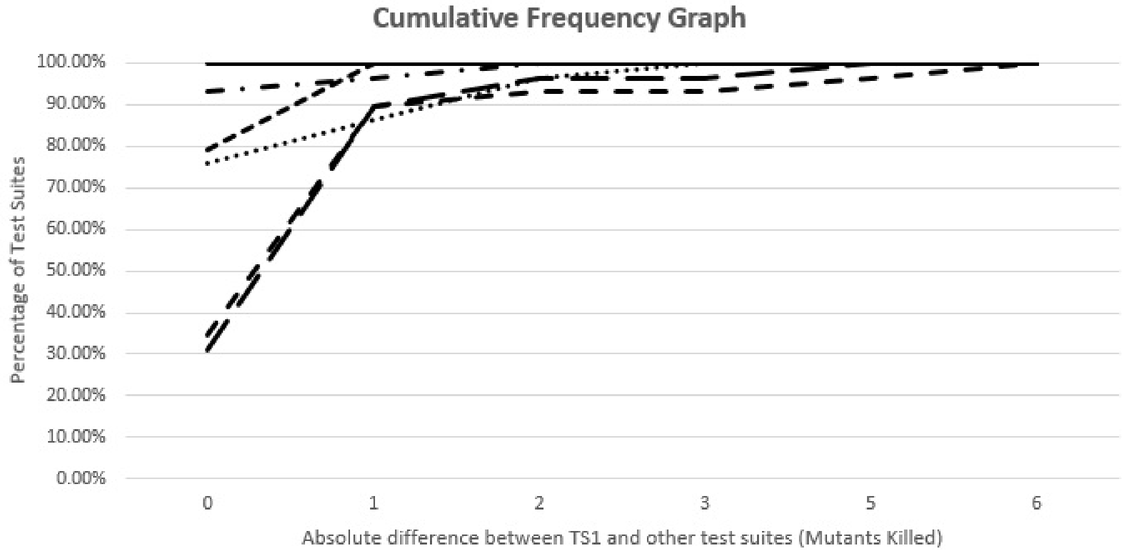

Figure 3.7: Cumulative Frequency Graph

Figure 3.7 is a cumulative frequency graph. A value on the X-Axis pertains to the absolute difference between TS1 and a given test suite, in terms of the number of killed mutants, and the Y-Axis represents a percentage of test suites. The intersection between the X-Axis and Y-Axis can be interpreted as follows: the percentage of test suites on the Y-Axis deviated from TS1 by either the number of mutants represented by the X-Axis or fewer. Each line on the graph corresponds to a Probabilistic IR.

It can also be observed in Figure 3.6, that Probabilistic IRs experience more variance than Deterministic IRs. Interestingly however, Figure 3.7 illustrates that the severity of these variations are relatively small. For example, it shows that, across all of the Probabilistic IRs, the minimum, average, and maximum number of test suites that only deviate from TS1 by one or fewer killed mutants, is 86.21%, 93.68%, and 100% respectively. This suggests that Probabilistic IRs perform very comparably across test suites.

3.3.5.2 False Positive Rate

Figure 3.8 is a scatter plot that shows the difference between TS1 and the other test suites, in terms of the number of false positives that were reported. The X-Axis corresponds to the difference in false positives between TS1 and the other test suites, and the Y-Axis is the number of test suites. The scatter plot shows that there was very little variance between TS1 and the other test suites. We conducted a Fisher's Exact Test to compare the false positive rate (measured in terms of the number of false positives and true negatives) of TS1 against the test suite that obtained the lowest false positive rate; we did not observe a significant difference. Similarly, we compared TS1 against the test suite with the highest false positive rate, measured in terms of their false positives and true negatives, and found that the difference was not statistically significant. These findings suggest that one's choice of test suite has a negligible impact on the false positive rate, and also provides a further indication about the generalisability of the results reported in Sections 3.3.2 to 3.3.4.3.

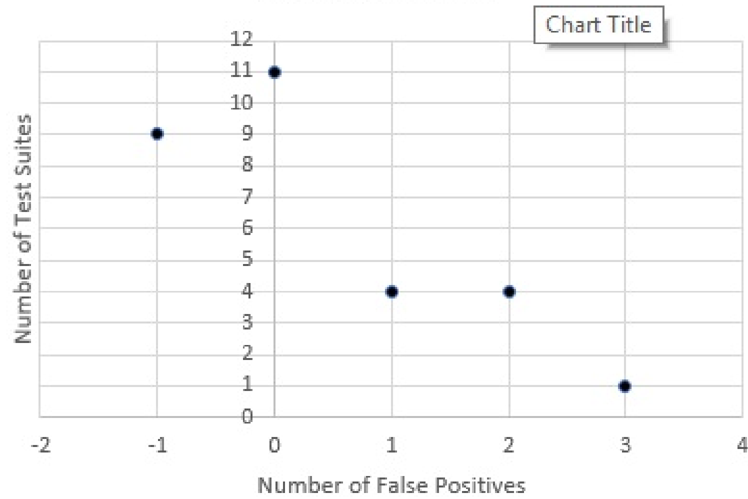

Figure 3.8: Difference between TS1's FPRs and the FPRs of TS2 to TS30.

3.3.5.3 Failure Detection Rate

As discussed in Section 3.3.2.2 the FDR is not applicable to Probabilistic IRs. Thus, only Deterministic IRs are considered in this section.

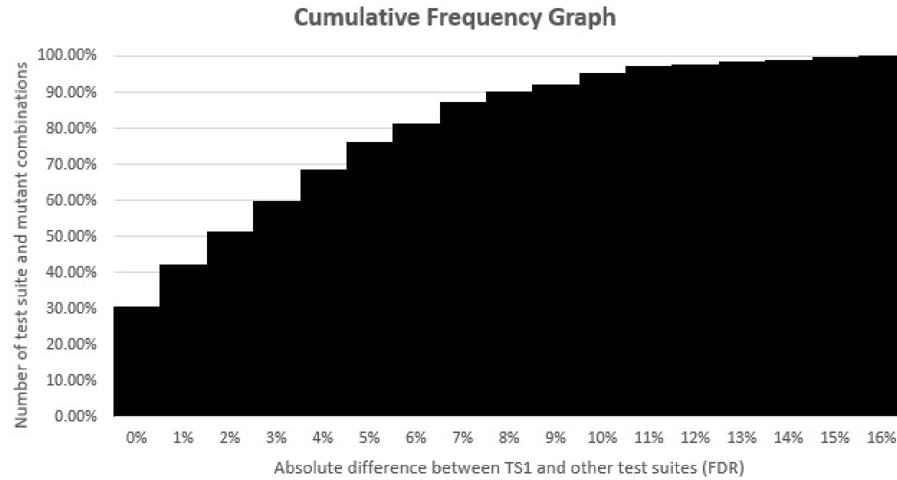

Figure 3.9: Cumulative Frequency Graph.

Let $\langle TS_i, m_j \rangle$ be a pair, such that TS_i denotes a particular test suite from TS2 to TS30, and m_j denotes a specific mutant. Let $Cases$ denote all such unique pairs. Figure 3.9 is a cumulative frequency graph. An interval of the X-Axis depicts the absolute difference in FDR between TS1 and a given test suite, and the Y-Axis corresponds to a percentage of $Cases$. The height of a bar illustrates the percentage of $Cases$ that have an FDR that differs from TS1 by a value that is either less than or equal to the value on the X-Axis interval associated with that bar. The graph indicates that there is relatively little difference between TS1 and the other test suites in terms of FDR e.g. 87.48% of $Cases$ only differ from TS1 by at most 7% FDR. We computed the average FDR of each mutant across TS2 to TS30, and compared these averages to the FDRs that were obtained by TS1 for these mutants

using a Mann-Whitney U Test[5]. The test revealed that there was no significant difference. These observations suggest that that test suite has little bearing on the FDR of the technique, and provides further evidence regarding the generalisability of the results that we reported across Sections 3.3.2 to 3.3.4.3.

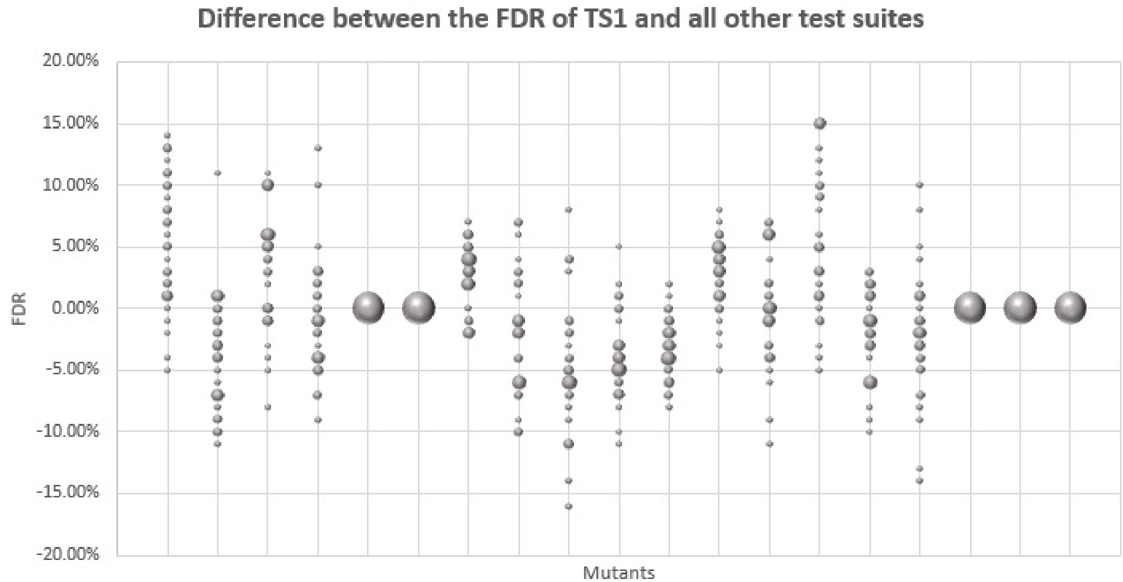

Figure 3.10: Difference between TS1's FDRs and the FDRs of TS2 to TS30.

Figure 3.10 is a bubble chart that shows the extent to which TS2 to TS30 differed from TS1 in terms of FDR, for each mutant. Each interval of the X-Axis corresponds to a specific mutant, and the Y-Axis expresses the difference in FDR between TS1 and the other test suites. The size of a bubble communicates the number of test suites from TS2 to TS30 that differ from TS1 to the same extent. For example, consider the mutant that is represented by the first interval of the X-Axis, and the bubble with the lowest value on the Y-Axis in this interval. This bubble demonstrates that one test suite obtained an FDR that was 5% lower than the FDR that was obtained by TS1, for the mutant represented by this interval. Figure 3.10 shows that the level of variance in terms of the FDR across test suites varies substantially for different mutants. This suggests that one's choice of test suite can be important for specific types of faults, but less so for others.

3.3.6 Discussion

3.3.6.1 A comparison of the effectiveness of Interlocutory Testing and traditional testing techniques

Let *Intentions* denote the user's intent for how the SUT should operate. Traditional testing techniques use the *Input* and *Intentions* to determine an expected outcome, *EOutcome*. The SUT is executed with the *Input* to produce an *Output*, and the *Output* is finally compared to *EOutcome*. The results of this comparison are used to determine the correctness of the SUT. Such techniques assume that the output accurately reflects the execution trace behaviours and thus the correctness of the former implies

[5]The Mann-Whitney U test statistic can be used to compare two groups based on one continuous measure [151]. It is a non-parametric technique and can therefore be used when the assumptions of parametric techniques are broken e.g. normal distribution [151].

the correctness of the latter. However, this assumption doesn't hold in the presence of coincidental correctness. Thus such techniques can be ineffective under these circumstances.

By contrast, Interlocutory Testing uses the *Input* in conjunction with the *Output* to determine *Intentions*. The SUT is executed with *Input* to produce an execution trace, $trace_i$, which is subsequently compared with *Intentions*. Again, the results of this comparison are used to determine the correctness of the SUT. Thus, Interlocutory Testing directly checks the correctness of execution trace behaviours. This means that Interlocutory Testing can operate effectively under the influence of coincidental correctness because it doesn't rely on the same unfounded assumption as traditional testing techniques.

Interlocutory Testing can check certain execution trace behaviours more precisely than other techniques because it checks these behaviours directly. For example, probabilistic IRs can test specific non-deterministic behaviours. By contrast, since other testing techniques base their evaluation on the output, they can only model the uncertainty introduced by non-determinism in general, into their expected test outcomes. This often means that they must weaken their thresholds for correctness [138]. For example, consider the Crossover Operator of a Genetic Algorithm; this operator generates a random number of random solutions and adds them to a pre-existing population of size PS. Since techniques like Assertions[6] can't predict the precise output, they must resort to testing with weak conditions e.g. $Population.size() >= PS$. This can increase their susceptibility to coincidental correctness because a larger number of outputs are deemed to be plausible, and thus there is more opportunity for coincidentally correct behaviours to map to a plausible output. It may also be necessary to prevent Assertions from being evaluated under certain conditions or in certain blocks of code, to counteract the effects of non-determinism [5]. This means that assertions may not even be able to test certain instances of such behaviours.

Consider the example in Section 3.1.1. The Crossover Operator's primary function is to expand a *Population* of size PS, such that $Population.size() \geq PS$. The exact number of additional individuals added to the *Population* is random. Since the Crossover Operator executes before the Selection Operator, it's entirely possible that the additional individual that was added to $Population_{SOI}$ by the fault, could have been generated by the Crossover Operator in a correct version of the system. Thus checking the integrity of $Population_{SOI}$ would not reveal the fault. This demonstrates that failures produced by some coincidentally correct faults can even be plausible in the first state they influence.

Coincidentally correct faults like these are plausible in all program states. Such faults are impossible to find using traditional testing techniques that are restricted to observing one program state. However, the example shows that it is possible to detect them by assessing the interplay between execution trace behaviours across multiple states. This is because a fault that appears plausible in one state, may not be plausible when considered in the context of multiple states simultaneously. IRs draw on multiple states simultaneously and so can detect such faults.

Other techniques can also check multiple states, but place restrictions on which states can be simultaneously considered. This can be detrimental to their fault detection capabilities for such faults. To illustrate, consider Assertions. Assertions can check multiple states, if state caching is feasible and

[6]Assertions are Boolean statements that are directly embedded in a system's source code [12]. If an assertion evaluates to true, then the test passes, otherwise the test fails [74].

is used [12]. Let $M1$ and $M2$ be two modules, such that a direct interface does not exist between these modules e.g. the Crossover and Selection components respectively. It has been reported that Assertions cannot correlate events between $M1$ and $M2$ [12]. This means it would not be possible for Assertions to consider states from Crossover and Selection simultaneously e.g. $CrossoverN$ and $Population_{soo}$, which means that Assertions cannot detect the coincidentally correct fault that was described in Section 3.1.1. Since IRs are not subject to such restrictions, their effectiveness is not compromised in this way.

Other testing techniques like assertions also use predicates. However, they're used in a different way. To illustrate, consider the logic, L, that selects a subset (*Parents*) of the *Population* to crossover, in a Crossover Operator. Predicates used by assertions can only be used to judge the correctness of logic that affects the outcome of these predicates. For example, an assertion may check $ParentsMod2 == 0$ and judge the correctness of L based on this assessment, because L affects this predicate. Predicates in Interlocutory Testing are not restricted in this way; an assessment of a predicate in Interlocutory Testing can imply the correctness of logic that does not affect the outcome of this assessment. For example, the ID associated with IOR_2 in Section 3.1.1.2 can report a failure in response to the fault outlined in Section 3.1.1, but the fault actually executes after all of the logic that was used to evaluate the ID has executed.

3.3.6.2 A comparison of the Usability of Interlocutory Testing and traditional testing techniques

The SUT is a set of program statements $SUT = \{s_1, s_2, ..., s_n\}$. Two program statements $s_i \in SUT$ and $s_o \in SUT$ are designated the input and output respectively. The user must develop an intuition into how s_i and s_o are related. s_i, s_o and this intuition form an IOR. This task is very intuitive, if one has in-depth domain knowledge. Chen et al. [45] used the Category-Choice Framework to develop an automated method of finding multiple groups of Input-Output pairs, such that interesting relationships may exist between these groups. Related analysis approaches may be useful for partially automating the exploration of relationships between inputs and outputs, and thus could simplify this task. For future work, we would like to investigate this possibility.

Having identified an IOR, the user must then identify execution trace behaviours that should manifest in executions in which this IOR is satisfied i.e. IDs. This task can be intuitive, if one has knowledge about the SUT's implementation details. It has been reported that such knowledge is typically readily available in software documentation like functional specifications [9]. Thus, a large proportion of this task may already be complete in many cases [9]. Several tools can also be used to partially automate the identification of such execution trace behaviours. For example, Program Slicing tools can be used to identify execution trace behaviours that either affect, or are affected by the outcome of an IOR [73]. One could narrow one's investigation of useful execution trace behaviours to those that were suggested by Program Slicing tools. Another example includes automated invariant detection tools like Daikon [74]. Such tools can be applied to executions in which the IOR is satisfied to identify potential relationships between intermediate state variables; an analysis of such relationships may expose interesting execution trace behaviours. Again, one could narrow one's investigation to such behaviours. These tools have been reported to be capable of reporting false positives [74] i.e. some of these relationships may be spurious. Thus, by using these tools, the user's task can be reduced

to checking the validity of relationships that have been proposed by these tools, and analysing valid relationships to expose potential execution trace behaviours.

Once multiple IORs have been identified and associated with IDs, the user can finally group IORs together and define relationships between them to form IRs. Such IORs tend to be highly related and so this task is typically very natural. For example, two IORs may collectively cover the entire input domain, but be mutually exclusive; thus, the user must define this mutual exclusivity.

The discussion above reveals that several tasks must be performed to obtain an IR. However, it also demonstrates that support tools are available that can partially automate the process. For future work, we would like to develop methods of increasing the degree of automation.

We believe that the effort involved in constructing IRs is comparable to other techniques. To illustrate this, we compare Interlocutory Testing with other popular techniques in terms of the assumed prerequisite skill-set of the user, the amount of implementation effort that is required, and the number of tasks that must be performed.

The discussion above reveals that the user is assumed to have domain knowledge and knowledge about the SUT's implementation details. Such a requirement is not uncommon for white box testing techniques e.g. Metamorphic Testing (see Section 2.3.2.1), Assertions [92] and some Model-based Testing techniques [140]. Promisingly, Interlocutory Testing requires less knowledge than some testing techniques. For example, since IDs can be expressed in any form, the user can select the means of describing the oracle that they are most comfortable with. Other techniques like Specification-based Testing restrict the user's avenues of expression to certain languages, some of which require an understanding of abstract mathematical concepts, that the user may not be familiar with [183].

With regards to the development task; an IR is comparably larger than some other types of oracles e.g. an assertion. However, one must note that Interlocutory Testing requires very few IRs to be effective (e.g. only 14 IRs were required for the Genetic Algorithm subject program — see Section 3.3.2.2, and only 3, 1, 1, and 4 IRs were used for the other subject programs — see Section 3.3.1), whilst other testing techniques like Assertions may require a large number of assertions [100]. We believe the proportionally higher cost of a single IR may be offset by the fact that fewer are required. Additionally, many other techniques are likely to involve substantially more programming effort than Interlocutory Testing. For example, N-version Testing may require the user to develop a second version of the SUT [81]. By comparison, Interlocutory Testing only requires the user to define associations among existing program entities — one does not have to implement these program entities [9].

Finally, Interlocutory Testing requires the user to perform several manual tasks to obtain an IR. Many other popular testing techniques also mandate numerous manual, labour intensive tasks. For example, Machine Learning based oracles may require the user to manually label training samples in a training dataset [64], select appropriate feature extractors [92], and write programs to translate data produced by the SUT into a compatible form for the ML algorithm (see Section 2.5.2.1).

3.4 Related Work

UCov is a tool that can assess the coverage adequacy of a test case in the context of regression testing [9, 8, 171]. Let Sys_{v1} be the SUT, and $Behav$ be an execution trace behaviour in Sys_{v1}. Suppose that a test case tc was generated, with the intention of exercising $Behav$. In UCov, this

intention is called a "Test Case Intention". UCov requires the tester to associate tc with $Behav$ to explicitly specify their Test Case Intention [9]. UCov can check whether $Behav$ manifests in Sys_{v1}, when executed with tc. Manifestation of $Behav$ indicates that the Test Case Intention of tc holds and thus, the coverage of tc is adequate and tc should be added to the test suite. The converse is true if $Behav$ does not manifest; the test case can be discarded in such situations. Suppose that the coverage of tc was deemed to be adequate. Further suppose that due to software maintenance, Sys_{v1} was subsequently modified into Sys_{v2}. $Behav$ may not manifest in Sys_{v2} with tc, thus the Test Case Intention of tc may not hold anymore and the coverage of tc may now be deemed to be inadequate. UCov can recognise this [9], and inform the tester, who may either modify the Test Case Intention of tc to realign it with Sys_{v2}, or discard tc.

The use of execution trace behaviours is central to both Interlocutory Testing and UCov [9]. However, the execution trace behaviours in each technique serve different purposes. In Interlocutory Testing, execution trace behaviours specify the intention of the SUT. Thus, checking whether these intentions hold is tantamount to checking the correctness of the SUT. To illustrate, let tc_c and tc_f be two test cases for Sys_{v1}, such that $Behav$ should manifest in both tc_c and tc_f, but does not manifest in tc_f because of a fault. Interlocutory Testing will detect the fault if Sys_{v1} is executed with tc_f, and if Interlocutory Testing predicts that $Behav$ will manifest. On the other hand, in UCov, an execution trace behaviour specifies the intention of a test case [8]. This means that if UCov was presented with tc_f and the aforementioned Test Case Intention (i.e. $Behav$ must be exercised by the test case), it would simply deem the coverage of tc_f to be inadequate, and the test case will be discarded. However, the converse is true if UCov is presented with tc_c. UCov's use of execution trace behaviours clearly prevents it from acting as an oracle for Sys_{v1}, however it does enable one to obtain a test case that will exercise $Behav$, and so it succeeds as a test case coverage adequacy criterion [9].

Let Sys denote the SUT. Sys^c_{loop} and Sys^f_{loop} are versions of Sys, in which a particular for loop iterates 50 and 49 times respectively, when these versions are executed with test case tc_{loop}. Sys^c_{loop} and Sys^f_{loop} are correct and faulty versions of Sys respectively. Let us suppose that $Sys \equiv Sys^f_{loop}$, was executed with tc_{loop}, and the fault was detected. The tester applies a bug fix to Sys, and thus Sys becomes $Sys \equiv Sys^c_{loop}$. After the bug fix has been applied, UCov expects the tester to associate tc_{loop} with execution trace behaviours that characterise the fault [8]. For example, in this case, tc_{loop} may be associated with a predicate that states that the for loop should iterate 50 times. This strategy guarantees coverage over areas of the code that have historically been faulty. Additionally, this strategy allows UCov to provide some limited oracle support. In particular, the exact same fault may resurface. This may happen because of an error in source control management that causes classes from Sys^f_{loop} to be used, instead of Sys^c_{loop} [8]. Under such circumstances, UCov will recognise that 50 iterations were not observed during the execution of tc_{loop}, and this can be used as an indication that the fault has resurfaced. Since the oracle that originally detected the fault will also detect the fault, we believe that this additional oracle support adds little value. The tester may insert a fault into the aforementioned bug fix, and UCov may be able to detect this [8]. For example, the loop may iterate 48 times because of a fault in the bug fix. Again, UCov will recognise that 50 iterations were not observed during the execution of tc_{loop}, and this can be taken as an indication of a fault in the bug fix.

Unlike Interlocutory Testing, the oracle support provided by UCov has the following severe limi-

tations. The discussion above demonstrates that UCov's oracle support is confined to the context of regression testing e.g. to a narrow range of faults that may exist in bug fixes — it cannot provide oracle support for the base source code of the system. In UCov, since an execution trace behaviour is associated with one specific test case [9], UCov can only use this execution trace behaviour as an oracle when executing this specific test case. Some faults may only result in failures in certain test cases. It is therefore possible for a fault to exist in a bug fix that does not manifest in this test case, but does result in a failure in other test cases. Such a fault may not be detected by UCov. Thus, the effectiveness of the oracle support that UCov can provide may also be limited. Finally, binding specific oracles to specific test cases can also limit the number of test cases that can be used in practice.

3.5 Threats to Validity

3.5.1 Internal Validity

There are several threats to internal validity. Firstly, several tools like MuJava, SPSS and Eclipse were used in this experiment. These tools may contain faults and thus reduce the validity of the experiment. However, since they are widely used and reputable, it's likely that most faults would have been accounted for.

Randomisation was used throughout the experiment where possible e.g. during test case generation, the SUT's execution and mutant selection. This served to reduce experimental bias.

To gauge the effectiveness of a testing technique accurately, it is necessary to ensure that it is exercised on a wide variety of fault types. To maximise the diversity of fault types in the experiment, we selected all of the available fault types in MuJava, and as discussed in Section 3.2.1.2, certain classes were excluded from mutation testing to also improve the diversity of the mutant sample. Unfortunately, MuJava could not generate all types of mutants for all classes. For example, in the Genetic Algorithm subject program, it could not produce any traditional mutants for the BinPackingAndCrossoverCommon class. This is a limitation of MuJava, and reduces the diversity of fault types within a class. However, fault types that were not available in one class, were available in others. For example, traditional mutants could be generated for BinPackingCrossover. Thus, this limitation is unlikely to have a substantial impact on the overall diversity of the mutant samples used in the experiment.

One of the mutation operators that is offered by MuJava is JSI. This mutation operator transforms a non-static variable into a static variable. Our experiments did not reset the state of variables that had been mutated by the JSI mutation operator to their default values at the end of a test case. This meant that the last state of these variables in one test case became the first state of these variables in the subsequent test case. In other words, in addition to transforming a non-static variable into a static variable, JSI also corrupted the initial state of this variable. We note that some of MuJava's other mutation operators can also corrupt the initial state of a variable e.g. the AOIU operator can transform a positive value to a negative value, during the initial assignment of a value to a variable. Thus, the additional consequence of using the JSI mutation operator is not dissimilar to other mutation operators. The JSI mutation operator was not used to produce many of the mutants that were included in our mutant samples e.g. it was only used to produce one mutant for the Genetic Algorithm subject program and was not used to produce any mutants for the other subject programs.

Thus, this redefinition of the JSI mutation operator only affected a negligible number of mutants.

Our method for determining whether a mutant was coincidentally correct, consisted of executing the mutant with a test suite, and applying a standard oracle to the output of each test case. This oracle checked every aspect of the output for correctness. If every output was deemed to be correct by this oracle, then our method concluded that the failures that were produced by the mutant did not propagated to the output, and thus the mutant was coincidentally correct. A mutant that is classified as coincidentally correct, based on this test suite, may not be classified as coincidentally correct for a different test suite. Thus, this is a threat to repeatability. However, this does not affect the validity of our results, because these classifications are correct in the testing context that our experiment was conducted in.

The presence of equivalent, infinite loop, and crashed mutants in the mutant sample could have been a confounding factor, because such mutants can skew empirical results. To eliminate this potential confounder, we manually inspected every mutant to identify equivalent mutants, and removed them from the sample. We also removed all infinite loop, and crashed mutants. Manual inspection is an extremely expensive task. It was therefore infeasible to generate an extremely large number of mutants. However, our sample is sufficiently large enough to obtain meaningful results.

Given that determining a mutant's equivalence to the original system is undecidable, and that manual inspection was used to perform this task, human error is possible [203]. We re-examined the 30 mutants that were used to address RQ5 to determine the extent to which misclassifications might have affected the validity of our results. We found that only two equivalent mutants had been misclassified as non-equivalent; these two misclassifications have led to a reduction in the mutation score. This suggests that the number of misclassifications that were made, was too small to have had a significant impact on the results.

Another issue was the imbalance of different IR groups e.g. 42 Deterministic vs 6 Probabilistic IRs. Since the SUT has a limited amount of logic that's amenable to certain groups, it was unfortunately impossible to gain a substantial and equal number of IRs for each group. It would have been possible to remove excess IRs from some groups to equalise the group sizes, but this could have led to a significant loss of valuable data.

Recall from Section 3.1.1.3 that the first phase of Interlocutory Testing involves capturing data about the execution trace, during the execution of the mutant, and that Interlocutory Testing can crash during this phase. In such cases, Interlocutory Testing has effectively detected the mutant. Our experiment did not distinguish between these crashes and system crashes, and so these mutants were conservatively removed from the experiment. Therefore, the experimental results presented in this chapter underestimate the technique's effectiveness. However, we do not believe that this had a significant impact on the results, since the technique already detects most of the mutants.

Three IRs had been partially developed for the Genetic Algorithm subject program. These IRs were not used in our experiments, but despite this, the logging function still captured execution trace data for these IRs. Certain mutants may potentially cause the logging function to crash while it is attempting to acquire this execution trace data. In such situations these IRs have effectively killed the mutant. We could not distinguish between these crashes and system crashes; thus, we had to conservatively remove these mutants from the experiment. Thus, our experimental results underestimate the technique's effectiveness; our estimate for the technique's effectiveness might have

been higher, had we leveraged these IRs, and been able to distinguish between different types of crashes. However, again, we do not believe this could have had a meaningful impact on the results, because most of the mutants were detected by the technique without these IRs.

Recall that a standard oracle was used to classify mutants as either coincidentally correct or standard. It might have been possible for certain mutants to cause this oracle to crash. In such situations, the oracle has effectively killed the mutant; thus the mutant should be classified as a standard fault. However, we did not distinguish between these crashes and system crashes, and so we conservatively removed such mutants. Our results demonstrated that Interlocutory Testing was able to kill 38/38 standard mutants, thus it is highly probable that the technique would have been able to detect these mutants. Thus, the most likely consequence of removing these mutants is that our results may underestimate the effectiveness of our technique. Again, this is unlikely to have had a meaningful impact on the results, because most of the mutants were detected.

Many logging tools are available. Different logging tools offer a distinct set of advantages and disadvantages. For example, some may have better performance, some may be more storage efficient, and some may offer greater flexibility than others. We developed our own logging tool, so that we could have control over the advantages and disadvantages that were offered by the tool. Let *Obj* be an object, that contains a private variable v, and a getter method $get(v)$ that returns v. *Obj* also contains another method $process()$ that uses v. Suppose that during the execution of a test case tc, the SUT invoked $process()$. Further suppose that $get(v)$ was never called during the execution, by the SUT. The evaluation of an IR, IR, may require v. Our logging tool is instrumented in the SUT and may execute parts of the SUT to obtain the required data. For example, the logging tool may execute $get(v)$ to obtain v. Thus, even though the code for the $get(v)$ method was not executed by the SUT, it was still executed by the logging tool. This demonstrates one advantage of our logging tool — it can increase code coverage. It is unclear how the technique might perform, when used in conjunction with a different logging tool. Thus a threat to repeatability might be one's choice of logging tool. For future work, we would like to investigate the impact of different logging tools on the technique.

The presence of real faults may confound the results i.e. a test case may result in failure, and one may believe this to be a consequence of a mutant, when in actuality, the failure may have originated from a real fault. To address this problem, the test suite was executed on a "correct" version of the SUT, and real faults were repaired before mutation testing was conducted.

It was infeasible to remove one of the real faults. To prevent the real fault from contaminating the experiment results, we modified IRs that were sensitive to the real fault to ensure that they could not detect this fault, and avoided mutants that included mutations to the real fault.

One of the relations that was sensitive to the real fault was DecidingWhoShouldMutate. This relation could only detect the real fault in an extremely rare set of circumstances. Unfortunately, we had not encountered these circumstances prior to conducting our experiments and so were unaware of its sensitivity to the real fault. As such, we neglected to modify the relation to remove its capability of detecting the real fault, and this initially led to the confounding of the results of RQ5. We cleaned these results by removing the DecidingWhoShouldMutate relation from them, and this mitigated the confounding factor. After the results of RQ5 had been cleaned, we did not have a single result set in which the DecidingWhoShouldMutate relation had reported a failure. This means that this IR could not have confounded any of our results.

3.5.2 External Validity

Five subject programs were used in our experiments. These subject programs varied in terms of size, problem domain, and susceptibility to coincidental correctness. Despite these differences, the technique obtained similar results across all of the subject programs. This suggests that the technique may operate effectively regardless of which system it's applied to, we therefore believe our results may generalise to a wide range of systems. However, we acknowledge that an investigation of the technique's effectiveness across more subject programs would be necessary to verify this conjecture. We therefore intend to study more subject programs in future work.

All of our subject programs were written in Java, and were object oriented. Thus, our experiments do not provide any indication about how Interlocutory Testing might perform for programs written in other languages e.g. C++, Prolog, or Ruby, or in other paradigms e.g. procedural. Additionally, our experiments only focused on functional testing, and so only demonstrates Interlocutory Testing's effectiveness in this context. It is conceivable that Interlocutory Testing might not be as effective in other contexts. For example, Interlocutory Testing may have little applicability for GUI Testing because there is unlikely to be many useful Interlocutory Decisions in this context. Additionally, Interlocutory Testing might be less useful for time sensitive applications, because instrumentation may reduce the performance of these systems. Again, we believe it would be useful to study more subject programs, to verify these conjectures, and gain further insights into the effectiveness of the technique.

We did not use code with known real faults in the experiment because there was not a substantial amount of data on real faults that had been encountered for the subjects. Regardless, had there been, their use could have biased the experiment since the author would be aware of the conditions that are necessary to detect them prior to IR design. Instead, MuJava was used to produce artificial faults. As previously discussed, research has shown that these are relatively accurate simulations of real faults and so should not reduce generalisability. Additionally, the technique was capable of detecting previously unknown real faults, which are representative and can't bias the experiment.

We adopted an iterative development process to develop our IRs. Thus, the design of each IR evolved throughout the development process. As such, an incarnation of an IR, IR_{old}, may have had IORs and/or IDs that are not used in a later incarnation of that IR, IR_{new}. Despite this, the logging function still captures the execution trace data that is required for the evaluation of these IORs and/or IDs and stores them in a log file, and Interlocutory Testing still extracts the execution trace data pertaining to these IORs and/or IDs from the log file (i.e. Phases 1 and 2 in Section 3.1.1.3), for IR_{new}. This means IR_{new} may have access to additional implicit IDs. We note that this is compliant with Interlocutory Testing's technique description, which states that relevant data is captured/extracted for the assessment of an IR's IORs and IDs, but does not rule out the potential for capturing/extracting data that is not relevant for the assessment of these IORs and IDs. Although one could remove these additional implicit IDs, we believe that testers would realistically elect to retain them, since they could improve the effectiveness of the testing process. Thus by keeping them, we have improved the representativeness of our experiments.

Appendices A, C, D, E, and F provide a brief summary of the main aspects of our IRs. These summaries detail the main checks that are performed by our IRs. For example, a summary of an IR might state that it checks that a list $L1$ is a permutation of another list $L2$. There are different

methods of implementing such a check. In continuation of the previous example; one could check the following five conditions: $L1$ and $L2$ are the same size ($Cond_1$), for each element in $L1$ there exists an equivalent element in $L2$ ($Cond_2$), and vice versa ($Cond_3$), and for each element in $L1$ that has more than one occurrences in $L1$, this element occurs the same number of times in $L2$ ($Cond_4$), and vice versa ($Cond_5$). Alternatively, one could implement this check by only verifying a subset of these conditions e.g. $Cond_1$, $Cond_2$, and $Cond_3$. One might do this to enhance the performance of the testing process, or increase the speed of the IR development process. Consider another example; let $Input = \{1, 2, ..., n\}$ be a list of integers. Suppose that a loop iterates over all elements of $Input$. Let m_i be the member of $Input$ being considered on iteration i, and $Prev_i$ denote the set of members that were considered on the iterations that executed prior to execution of iteration i. Further suppose that on each iteration i, the loop stores m_i in another list $Output$. One check that an IR might perform may include verifying whether the member of the population being considered on a certain iteration had not been considered before. One method of implementing this check might include the following: for a given iteration i, check whether m_i exists in $Prev_i$. Alternatively, one could check whether $Input$ is a permutation of $Output$. Thus, one threat to repeatability might include variations in implementation details.

The results demonstrated that different IRs obtained different levels of effectiveness. Thus, the effectiveness of the technique may vary considerably, depending on one's choice of IRs. This may be a threat to generalisability.

A large number of test case generation strategies are available e.g. branch coverage, statement coverage, and def-use pair coverage. We only used one strategy for test case generation in our experiments, which was random generation. This strategy is beneficial because it reduces experimental bias. The technique might perform differently, depending on which test case generation strategy is used, but since our experiments do not explore other test case generation strategies, it is unclear how differently the technique might perform. Thus, this might be a threat to generalisability, and a future research avenue.

Recall that our test case generation strategies involved generating random values for our test cases. As discussed in Section 3.2, these strategies constrained the selection of allowable values for our variables, to prevent the generation of infeasible test cases. They were also engineered to be more likely to select smaller values for some of these variables. This was beneficial because it reduced the average cost of running test cases e.g. fewer generations for our Genetic Algorithm subject program. One threat to repeatability might be that other's may deploy different test case generation strategies.

3.5.3 Construct Validity

Several metrics have been shown to be accurate measures of effectiveness and are widely used by the testing community e.g. MS, FDR and False Positive Rate. This chapter has only used such metrics.

3.5.4 Statistical Validity

Non-parametric statistics were used in the place of parametric statistics, when the assumptions made by parametric statistics did not hold e.g. normal distribution.

More precise statistical techniques than those that were used in this chapter may have been available. However, the statistical techniques used here are ubiquitous and are considered to be robust.

There is a trade-off associated with the significance threshold of a statistical test. In particular, if one decreases the significance threshold, this will lead to an increase in the chance of incorrectly accepting the null hypothesis. Conversely, if one were to increase the significance threshold, then the chance of incorrectly rejecting the null hypothesis also increases. We used 0.05 as the significance threshold in every statistical test that was conducted, because we felt that this choice of significance level offers a reasonable trade-off.

3.6 Conclusion

In this chapter, we explored coincidental correctness; a widespread issue in software testing where the SUT malfunctions, but the output is plausible. Interlocutory Testing was introduced to resolve this issue. The feasibility of this technique was established. The experimental results demonstrated that Interlocutory Testing is effective, obtaining a mutation score of 87%, detecting 1 false positive and 12 real faults. It was able to detect coincidentally correct and standard faults in a non-testable SUT. This is promising and suggests that Interlocutory Testing might be an effective solution for coincidental correctness. These results indicate that Objective 1 has been accomplished (see Section 1.1). It also suggests that it might be a useful technique for non-testable systems in general.

We also investigated whether these results could generalise to other systems. We replicated the experiment across four subject programs that varied in terms of size and problem domain. Interlocutory Testing was capable of detecting 39/40 fault across these subjects, found three real faults and reported 0 false positives. The consistency of these results with our Genetic Algorithm subject program suggests that our results can generalise to other systems.

Similarly, we explored whether our results could generalise to other test suites by analysing the consistency of the technique's effectiveness across 30 different test suites. We observed that the technique performed very consistently across these test suites, in terms of the mutation score, failure detection rate, and the false positive rate. These observations indicate that our results can generalise to different test suites.

We also determined the minimum number of IRs required to obtain maximum effectiveness — 14 were sufficient for the GA subject program. Additionally, we observed that only 4, 1, 1, and 3 IRs were used for the other subject programs. This demonstrates that relatively little test effort is required to realise Interlocutory Testing's full potential, assuming that the relations are appropriately designed. We additionally explored how various characteristics of an IR relate to its effectiveness e.g. the most effective relations have very context sensitive IORs; these preliminary findings provide some insight into how one might develop a small set of effective IRs e.g. developing IRs for highly coupled areas of the system. However, these findings do not constitute a comprehensive set of guidelines. For future work, we believe it would be beneficial to develop such a set of guidelines.

It would also be beneficial to explore the usability of Interlocutory Testing further. In particular, we would like to devise a partially automated method of generating IRs. We are aware of promising research that attempts to partially automate the development of oracles, and we believe some of this research may be useful for Interlocutory Testing. For example, Chen et al. [45] developed an automated method of finding potentially useful Input-Output pairs for Metamorphic Testing based on the Category-Choice Framework. We believe that similar technologies may also be useful for the

IOR identification process.

Recall that the typical false positive rate of a Probabilistic IR is currently either extrapolated from empirical data, based on the tester's expertise, or based on an analysis of the randomised properties of the SUT. Situations may exist in which empirical data doesn't exist and the tester may not have any relevant expertise. In such situations, one must analyse the randomised properties of the SUT, which can be a difficult task. Thus, simpler methods for determining the typical false positive rate, or a completely different approach that does not use the typical false positive rate warrant investigation. An example of the latter might incorporate the Holm–Bonferroni method.

Finally, we would like to explore the viability of an IR centric test adequacy criterion. Such a criterion would deem a test suite to have adequate coverage if each IOR that is associated with each IR is evaluated at least once by the test suite. Interlocutory Testing may be more effective, if it was used in conjunction with such a test suite.

In summary, this chapter introduced Interlocutory Testing, a testing technique that can operate effectively in the presence of coincidental correctness, and fulfilled Objective 1 of the thesis. The next chapter modifies Metamorphic Testing, by means of combining it with Interlocutory Testing, to reduce the susceptibility of Metamorphic Testing to coincidental correctness.

Chapter 4

Interlocutory Metamorphic Testing

Chapter 2 revealed that Metamorphic Testing (MT) is the most widely studied testing technique, in the context of the oracle problem. It also presented some evidence that indicated that the effectiveness of this technique can be compromised by coincidental correctness, and that coincidental correctness is prevalent. We therefore believe that research that alleviates the impact of coincidental correctness on Metamorphic Testing, could add substantial value. Chapter 3 introduced a new testing technique called Interlocutory Testing, that can operate effectively in the presence of coincidental correctness. The motivation for the work described in this chapter was the potential for extending MT with Interlocutory Testing to enable MT to operate effectively in the presence of coincidental correctness. Thus, the research that was undertaken in this chapter attempts to address Objective 2 (see Section 1.1). In this chapter, we present an integrated approach called Interlocutory Metamorphic Testing (IMT), that combines these two techniques, and investigate its prowess in coincidentally correct testing scenarios.

In summary, this chapter makes the following contributions:

- Further evidence that suggests that MT is ineffective in the presence of coincidental correctness.
- A new testing technique called Interlocutory Metamorphic Testing.
- A set of test oracles based on Interlocutory Metamorphic Testing.
- Case studies that investigate the feasibility and effectiveness of Interlocutory Metamorphic Testing.

All of the relevant background material for this chapter can be found in Chapters 2 and 3. The structure for this chapter is as follows. IMT is introduced in Section 4.1. We conducted several case studies that investigate the feasibility and effectiveness of IMT. The experimental design for these case studies can be found in Section 4.2, the results of these experiments are presented and discussed in Section 4.3, and factors that may threaten the validity of these results are considered in Section 4.4. Finally, conclusions are drawn in Section 4.5.

4.1 Interlocutory Metamorphic Testing — Technique Description

This section leverages the Tournament Selection Operator (TSO) example from Section 3.1.2 as a running example. To briefly recap the key elements of the example; the input of TSO is a population. A tournament is a subset of this population. TSO employs an iterative process, in which on a given iteration, a tournament is randomly formed, and one solution in this tournament is randomly designated the winner. The random selection of a winner is biased towards solutions with greater fitness values. Let $OutputPopulation$ denote the set of winners across all of these tournaments. The termination condition for the iterative process is $OutputPopulation.size() == PS$, where PS denotes the maximum population size.

4.1.1 Intuition

As discussed above, the effectiveness of Metamorphic Testing (which was introduced in Section 2.3) can be negatively affected by coincidental correctness, and Interlocutory Testing (which was introduced in Section 3.1) can alleviate coincidental correctness. We reasoned that the integration of Interlocutory Testing into Metamorphic Testing might ameliorate the negative effects of coincidental correctness on Metamorphic Testing.

Recall that in Metamorphic Testing, an MR has a Metamorphic Test Group (MTG) that consists of source and follow up test cases, and that these test cases are executed to enable the MR to verify whether an expected property (Metamorphic Property) between these test cases holds. One method of integrating Interlocutory Testing into Metamorphic Testing involves recording the execution traces of these source and follow up test cases, and evaluating these execution traces with IRs. The intuition behind Interlocutory Metamorphic Testing is to combine Interlocutory Testing and Metamorphic Testing in this manner, with the expectation that this will reduce the effects of coincidental correctness on Metamorphic Testing.

As an aside, other researchers have adopted similar approaches for merging Assertions [174] and Machine Learning Oracles [26] into Metamorphic Testing. Our method of integration was inspired by these approaches.

4.1.2 Interlocutory Metamorphic Testing

In this section, we provide a detailed explanation of how Interlocutory Metamorphic Testing realises the intuition above.

Algorithm 5: Interlocutory Metamorphic Relation

Input: The execution trace log file *LOG* of a single test case *tc* from the test suite *ts*

Output: $MRVerdicts$, $IRVerdicts$

1 Let $MRVerdicts$ be an empty list;

2 Let MR_{ET} be an empty set;

3 $MetRel$ denotes the Metamorphic Relation that is associated with this IMR. Let *RelevantLOG* be the set of all subsequences in *LOG* that could be used as a source test case by $MetRel$;

4 **foreach** *subsequence* $MRET_i$ *in RelevantLOG* **do**

5     $MRET_i$ is used to generate additional source and follow up test cases, which are then executed to produced additional execution traces. Let $AdditionalET_i$ denote this set of additional execution traces;

6     $MRVerdict = MetRel.assessMR(MRET_i, AdditionalET_i)$;

7     $MRVerdicts.add(MRVerdict)$;

8     $MR_{ET}.add(\langle MRET_i, AdditionalET_i\rangle)$;

9 **end**

10 Let $IRVerdicts$ be an empty list;

11 **foreach** *execution trace* ET_j *in* MR_{ET} **do**

12   The IRs that are associated with this IMR are evaluated based on ET_j, using the methodology outlined in Section 3.1.1.2. The outcomes of these evaluations are stored in $IRVerdicts$;

13 **end**

An oracle in IMT is called an Interlocutory Metamorphic Relation (IMR). Algorithm 5 outlines the procedure for evaluating an IMR for a single test case. In this section, we first explain this procedure, and then describe how the output of this procedure should be interpreted.

Let MR be a Metamorphic Relation for TSO. The source test case of MR is $select(P_1)$, where the function $select()$ executes TSO, P_1 is a population, and the output of $select(P_1)$ is a version of P_1 that has been subjected to tournament selection. Similarly, the follow-up test case associated with MR is $select(P_2)$, where P_2 is a superset of P_1 that contains one additional member. MR checks that the output populations of $select(P_1)$ and $select(P_2)$ have the same size. An IMR contains one MR e.g. MR.

When the SUT is executed with a test case, an execution trace *LOG* is produced. *LOG* serves as input into Algorithm 5. Some subsequences of *LOG* can be used as source test cases by IMR's MR. For example, there may be multiple subsequences of the execution trace that each capture information about one invocation of TSO; any one of these subsequences can be used as a source test case for MR. Line 3 of Algorithm 5 defines *RelevantLOG* to be the set of all such subsequences.

Let $MRET$ be a subsequence in *RelevantLOG*. Line 5 of Algorithm 5 uses $MRET$ to generate additional source and follow up test cases; in the case of MR, this would involve extracting P_1 from $MRET$ and adding an additional random member to it to obtain P_2 to produce one fol-

low up test case — $select(P_2)$. Line 5 also executes these additional test cases to produce additional execution traces. Let $AdditionalET$ denote these execution traces. Line 6 then leverages $MRET$ and $AdditionalET$ to assess the MR, Lines 1 and 7 record the MR's SUTPossiblyCorrect/SUTFaulty verdict in $MRVerdicts$, and Lines 2 and 8 store $MRET$ and $AdditionalET$ as a pair $\langle MRET, AdditionalET \rangle$ in MR_{ET}. This process is repeated (by means of Line 4) for each subsequence in $RelevantLOG$.

For a given pair $\langle MRET, AdditionalET \rangle \in MR_{ET}$, ET_j can either be $MRET$ or be a member of $AdditionalET$. An IMR is associated with a set of IRs, such that an evaluation of an IR from this set can be conducted based only on execution trace data in ET_j. To illustrate, TournamentPIR, which was discussed in Section 3.1.2.1 only uses information that is available in a single invocation of TSO, and so can be associated with an IMR that contains MR. However, such an IMR *cannot* be associated with the IR discussed in Section 3.1.1.2 because this IR requires information from the Crossover Operator, which is not available in ET_j. Lines 10 – 12 carry out the following operations: the IRs that are associated with the IMR are evaluated (using the methodology that was outlined in Section 3.1.1.2) based on each ET_j in MR_{ET}, and the outcome of these evaluations are stored in $IRVerdicts$. The output of Algorithm 5 is $MRVerdicts$ and $IRVerdicts$.

The output should be interpreted as follows: The IMR has reported SUTFaulty if $MRVerdicts$ contains at least one SUTFaulty verdict, or if $IRVerdicts$ contains at least one SUTFaulty verdict from a Deterministic IR.

The procedure detailed in Algorithm 5 is sufficient for IMRs that only contain Deterministic IRs. However, an additional step is required, if the IMR contains probabilistic IRs. This step involves using a modified version of the procedure outlined in Section 3.1.2 for each Probabilistic IR associated with the IMR. In IMT's version of the procedure, $count(SUTFaulty_{tc_i})$ and $count(SUTPossiblyCorrect_{tc_i})$ are counts of the SUTFaulty and SUTPossiblyCorrect verdicts in $IRVerdicts$ respectively, for the Probabilistic IR under consideration. The IMR is said to have reported SUTFaulty if this modified procedure concludes SUTFaulty for any of the Probabilistic IRs.

In summary, an IMR reports a failure when at least one of its IRs or its MR fails. We envision that one would use multiple IMRs in practice. Some IMRs may fail, and some may not. This should be interpreted as follows: the SUT should be considered to be faulty if at least one IMR reports a failure, and should be deemed to be correctly implemented if none of the IMRs fail.

4.2 Experimental Design

This section outlines our research questions and describes the experiments that were conducted to answer them.

4.2.1 Research Questions

RQ1 Is Interlocutory Metamorphic Testing a feasible[1] testing technique? This research question assesses the feasibility of Interlocutory Metamorphic Testing by exercising it on a variety of widely used programs.

[1] In the context of this research question, feasibility refers to whether the technique is capable of carrying out its designated task.

RQ2 How effective is Interlocutory Metamorphic Testing in the presence of coincidental correctness? The primary objective of IMT is to extend the capability of MT to situations where coincidental correctness is present. We therefore conducted experiments to measure its effectiveness under these conditions.

RQ3 What impact does Interlocutory Metamorphic Testing have on the effectiveness of Interlocutory Testing? The integration of Interlocutory Testing and MT will have had some influence on the effectiveness of the underlying techniques. The impact on MT is covered by RQ2. This research question investigates how the effectiveness of Interlocutory Testing is affected by this union.

RQ4 What effect does the test suite have on the effectiveness of IMT? As discussed in Section 4.1, IMT generates additional test cases, which is an important aspect of the technique. Since these additional test cases are generated based on the test suite, it's possible that the test suite can influence the effectiveness of IMT. This research question investigates this possibility.

4.2.2 Subject Programs

This chapter leverages the same five subject programs that were used in Chapter 3. A description of the four subject programs that were used to answer RQ1 (Dijkstra's Algorithm, Bubble Sort, Binary Search, and Knuth-Morris-Pratt), and the subject program that was utilised for RQ2 — RQ4 (Genetic Algorithm for the Bin Packing Problem) can be found in Section 3.2.3.1 and Section 3.2.1.1 respectively. Our reasons for selecting these subject programs are the same as those that were outlined in Section 3.2.3.1 and Section 3.2.1.1.

4.2.3 Test Cases

We generated 400 test cases across the four subject programs that were used to answer RQ1 — 100 test cases were generated per subject program. Descriptions of the test case generation strategies that were used for each of these subject programs can be found in Section 3.2.3.3.

The subject program that was used for RQ2 — RQ4, has two test suites. The first test suite, which consists of 100 test cases and is referred to as TS1, is the same test suite that was used in Chapter 3. The second test suite also consists of 100 test cases, and was generated using the test case generation strategy that was outlined in Section 3.2.1.3. We refer to this test suite as TS2. Sections 4.3.2 to 4.3.3.2 use TS1 to answer RQ2 and RQ3. TS1 and TS2 are used in conjunction with each other to answer RQ4 in Section 4.3.4.

Our justifications for the use of these test case generation strategies and TS1 can be found in Sections 3.2.3.3 and 3.2.1.3.

4.2.4 Faults

We generated 120 mutants across all five of the subject programs that were listed in Section 4.2.2, using the same mutant generation strategies that were discussed in Sections 3.2.1.2 and 3.2.3.2. In particular, a total of 40 mutants were generated across the four subject programs that were used to answer RQ1 — 10 mutants per program. 80 mutants were generated for the subject program that was used to answer RQ2 — RQ4; 48 mutants were classified as coincidentally correct and the remaining 32

were labelled as standard. The same justifications that were presented in Sections 3.2.1.2 and 3.2.3.2 apply to the decisions made in this section.

4.2.5 Interlocutory Metamorphic Relations

In this chapter, we use the same IRs that were used in Chapter 3. The IRs are listed in Appendices A, C, D, E, and F for the Genetic Algorithm for the Bin Packing Problem, Dijkstra's Algorithm, Bubble Sort, Binary Search and Knuth-Morris-Pratt subject programs respectively. Each IR is also accompanied by a description of the main aspects of that IR, as well as a unique identifier. These groups of IRs were selected because they were large enough to support the various analyses that they were used for in this chapter. Sections 4.2.5.1 to 4.2.5.5 outline the IMRs that were used in this chapter. Each IMR, IMR, is described by a pair $\langle MR, IRs \rangle$ such that MR and IRs is the MR and the set of IRs that are associated with IMR respectively. The IRs in this section, are represented by their unique identifiers. It has been reported that a small number of MRs is typically sufficient [109]. Thus, we limited our experiments to a small number of IMRs to increase the representativeness of the experiments.

4.2.5.1 Dijkstra's Algorithm

- **IMR1**
  - **MR:** Let $Dijkstra(G, S, E)$ be an implementation of Dijkstra's Algorithm that accepts a graph, G, Start Node, S, and End Node, E, as input. The total weight of the path produced by $Dijkstra(G, S, E)$ should be the same as the total weight of the path produced by $Dijkstra(G, E, S)$.
  - **IRs:** 1 - 4

4.2.5.2 Bubble Sort

- **IMR1**
  - **MR:** Let $Reverse(I)$ be a function that reverses the order of a list, I. The output of *BubbleSort*(I) should be the same as *BubbleSort*(*Reverse*(I)).
  - **IRs:** 1

4.2.5.3 Binary Search

- **IMR1**
  - **MR:** Let *Item* be the item being searched for in *List*. If the first element of *List* is *Item*, then incrementing the value of the last element of *List* should not change the subject program's output. Otherwise, decrementing the value of the first element of the list should not change the subject program's output.
  - **IRs:** 1

4.2.5.4 Knuth-Morris-Pratt

- **IMR1**
  - **MR:** Let $PATTERN$ be the pattern being searched for in $TEXT$. Additionally, let $PATTERN2$ be a prefix of $PATTERN$, such that $PATTERN.size() - PATTERN2.size() = 1$. If the subject program can find $PATTERN$ in $TEXT$, then it should also be able to find $PATTERN2$ in $TEXT$.
  - **IRs:** 1 - 3

4.2.5.5 Genetic Algorithm for the Bin Packing Problem

Recall that a real fault is present in the subject program. One of the steps taken to remove the possibility of this real fault affecting the validity of our results was ensuring that the MRs and IRs were not sensitive to the real fault.

- **IMR1**
  - **MR:** Let *InitialPopulationSize* be an input parameter value denoting the intended population size and *GeneratePopulation*(I) be a method that creates a random population of size I. *GeneratePopulation*(*InitialPopulationSize*+1).*size*()−*GeneratePopulation*(*InitialPopulationSize*).*size*() == 1.
  - **IRs:** 1 - 5
- **IMR2**
  - **MR:** Let P denote a population and *adapt*(X) be a method that accepts a population as input and replaces one random member with a new randomly generated member. *generateNextPopulation*(Y) is a controller method that applies crossover, mutation and selection to a population Y. *generateNextPopulation*(P).*Size*() == *generateNextPopulation*(*adapt*(P)).*Size*().
  - **IRs:** 6, 7, 9, 10
- **IMR3**
  - **MR:** Let P_i represent the input population and P_p be a permutation of P_i. Furthermore, let $Crossover(P_i)$ and $Crossover(P_p)$ be the products of applying the crossover operator to P_i and P_p respectively. Although P_i and P_p contain exactly the same elements, the crossover operator may transform these populations in different ways, thus $Crossover(P_i)$ may not be a permutation of $Crossover(P_p)$. However, since the nature of this transformation is the addition and not removal of individuals, the following conditions should hold: P_i and P_p should both be subsets of $Crossover(P_i)$ and $Crossover(P_p)$.
  - **IRs:** 15 - 19, 21 - 40
- **IMR4**
  - **MR:** Let P be the population, $mutate(Y)$ be a mutation operator that can be applied to a population and *createRandomPopulation*(I) be a function that creates a random

population of size I. $mutate(P).size()*2 == mutate(createRandomPopulation(P.size()).addAll(P)).size()$.

  - **IRs:** 31 - 48

- **IMR5**

  - **MR:** Let P_1 and P_2 be two populations, such that P_1 is a subset of P_2 and $P_2.size() - P_1.size() == 1$. Also, let $select(P_i, PSize)$ denote a population that has been derived from the application of the selection operator to a population P_i, based on a genetic algorithm's population size parameter $PSize$. $select(P_1, PSize).size() == select(P_2, PSize).size()$.
  - **IRs:** 11 - 14

4.3 Results and Discussion

In this section, we present the results obtained from our experiments and use them to address the research questions that were outlined in Section 4.2.

4.3.1 RQ1. Is Interlocutory Metamorphic Testing a feasible testing technique?

To answer RQ1, we leveraged the following four subject programs: Dijkstra's Algorithm, Bubble Sort, Binary Search, and Knuth-Morris-Pratt. Each of these subject programs was associated with 100 test cases. MuJava was used to generate 10 mutants per subject program, thus a total of 40 mutants were used across these subject programs. IMT successfully killed all of the mutants, thereby obtaining a mutation score of 100%. This indicates that IMT is a feasible testing approach. The consistency of the results across these subject programs also suggests that our findings regarding its feasibility may be generalisable.

The Failure Detection Rate (FDR) is a measure of the proportion of test cases that detect a fault. We also observed that the average FDR for Dijkstra's Algorithm, Bubble Sort, Binary Search, and Knuth-Morris-Pratt were 100%, 100%, 94.8%, and 78.3% respectively. The FDR fluctuates substantially, which means that the effectiveness of the technique varied for different subject programs. This indicates that even though the technique is feasible, one can expect differing levels of effectiveness for different programs. Promisingly, however, the lowest FDR was 78.3% which is relatively high, and this could indicate that the support it provides for programs that it is less effective for may be acceptable.

4.3.2 RQ2. How effective is Interlocutory Metamorphic Testing in the presence of coincidental correctness?

In this section, we explore RQ2. This research question is particularly important because it focuses on IMT's ability to achieve its primary objective — extending MT to scenarios involving coincidental correctness. To achieve this, we use the Genetic Algorithm subject program, test suite TS1, and 80 mutants.

Mutants	All IMRs	IMR1	IMR2	IMR3	IMR4	IMR5
All	78.75%	11.25%	23.75%	40.00%	48.75%	7.50%
Standard	100.00%	15.63%	25.00%	75.00%	71.88%	0.00%
Coincidentally Correct	64.58%	8.33%	22.92%	16.67%	33.33%	12.50%

Table 4.1: IMT's Mutation Scores

	IMR1	IMR2	IMR3	IMR4	IMR5
IMR1		$p>0.05$	$p<0.05$	$p<0.05$	$p>0.05$
IMR2			$p>0.05$	$p<0.05$	$p<0.05$
IMR3				$p>0.05$	$p<0.05$
IMR4					$p<0.05$
IMR5					

Table 4.2: Pairwise comparisons between each IMR, based on Fisher's Exact Test

4.3.2.1 Mutation Score

Table 4.1 depicts the mutation scores obtained by our IMRs. The results are promising; they demonstrate that the technique was able to obtain an MS of 78.75%; it detected 32/32 standard faults and 31/48 coincidentally correct faults. When applied using MT, the MRs associated with these IMRs could not detect any of the coincidentally correct faults. This supports the observations made by others, that MT is ineffective in the presence of coincidental correctness, and demonstrates that IMT substantially improves the performance of MT for these faults. Thus, IMT has achieved its primary objective. Interestingly, these MRs could not detect any of the standard faults either. This indicates that the improvements offered by IMT generalise to other fault types.

MRs have limited potential with respect to the diversity of fault types they can detect. For example, the MR for IMR5 can only detect faults that cause the selection operator to remove an incorrect, and inconsistent (across executions), number of members from the population. MT performed poorly because the mutant sample did not contain the apposite faults for these MRs. Recall that crashed mutants were removed from the mutant sample; it is possible that some of these mutants may have been detectable by our MRs, which may partly explain why the mutant sample did not contain the appropriate faults for MT. Regardless, we believe that this did not affect the validity of our results because IMT would also have been able to detect these faults, so the comparison would have yielded similar results.

The effectiveness of each IMR (as measured by the number of killed and survived classifications) was compared to the effectiveness of all other IMRs, using multiple Fisher's Exact Tests. Table 4.2 shows the outcome of these comparisons, after the Benjamini-Hochberg correction has been applied. Table 4.1 indicates that some IMRs are more effective than others and Table 4.2 shows that the differences in effectiveness in many cases are statistically significant. This means that the effectiveness of IMT may vary depending on which IMRs are used. For example, if we had only used IMR3, the total MS would have been 40%.

Even though some IMRs are generally more effective than others, Table 4.1 also shows that in some cases, these IMRs may be outperformed by less effective IMRs for certain fault types. To illustrate,

Fault Types	**IMR1**	**IMR2**	**IMR3**	**IMR4**	**IMR5**
Standard	0	8	1	0	0
Coincidentally Correct	2	2	5	4	6

Table 4.3: Number of unique faults found by IMR, broken down by fault type

IMR1 obtains an MS of 11.25%, which is higher than IMR5; IMR5 only obtains an MS of 7.5%. IMR1 can detect 15.63% of the standard faults, whilst IMR5 cannot detect any of these faults. However, IMR5 obtains an MS of 12.5% for coincidentally correct faults, compared to just 8.33% for IMR1. Three such cases can be observed — IMR1 and IMR5, IMR2 and IMR3, and IMR3 and IMR4. For each of these cases, we compared the IMRs in terms of the number of killed and survived classifications, based on the fault type for which the IMR with greater effectiveness had been outperformed. None of the comparisons yielded a significant difference (Fisher's Exact Test: $p > 0.05$).

Table 4.3 shows the number of faults that were detected by exactly one IMR, and the IMRs that detected such faults. For ease of reference, we call such faults "unique faults". Every IMR finds unique faults and thus adds value, despite their overall effectiveness. This suggests that it may be sensible to leverage multiple IMRs, because this may increase the diversity of faults that are detectable. The table also shows that most of these unique faults are coincidentally correct; 9 faults in total are standard and 19 are coincidentally correct. This is not surprising because an IR's effectiveness for coincidentally correct faults is partly determined by the dispersion of program states in the execution trace that it considers during its evaluation (see Section 3.3.6.1). As discussed above, an IR in IMT can only consider program states that manifest from lines of code in $MRET$ and $AdditionalET$. This restriction means that the dispersion of program states that can be considered by IRs in each IMR is different; hence they are likely to find different coincidentally correct faults. A greater degree of overlap for standard faults is expected, since failures resulting from these faults can easily propagate throughout the SUT.

4.3.2.2 Failure Detection Rate

The previous section demonstrated that Interlocutory Metamorphic Testing can extend Metamorphic Testing's fault detection capabilities to a wider range of fault types. However, this is only one facet of effectiveness; another is the likelihood of detecting a fault. We measured this by calculating the FDR for all of the mutants that had been detected by deterministic IRs. Since the probabilistic IRs in IMT leverage all test cases to produce one verdict, the FDR measure is inapplicable to IMT when it incorporates probabilistic IRs. We therefore excluded probabilistic IRs from this analysis.

The average FDR is 50.8% across all faults, 46.4% for standard faults and 55.8% for coincidentally correct faults. This is relatively high and suggests that if IMT can detect a fault, it is likely to do so. Interestingly, the difference in IMT's FDR for standard and coincidentally correct faults is not statistically significant (Mann-Whitney U: $p > 0.05$). This suggests that IMT's effectiveness, in terms of FDR, is consistent across fault types. The minimum, maximum, skewness, and kurtosis of the FDR were 1%, 100%, 0.04, and 0.09 respectively. This demonstrates that IMT's FDR can vary substantially for different faults — IMT is more likely to detect certain faults in comparison to others.

IR Types	IMT	IT
All IRs	78.75%	82.50%
Deterministic IRs	56.25%	65.00%
Probabilistic IRs	63.75%	52.50%

Table 4.4: Overview of Interlocutory Testing and IMT's results

4.3.2.3 False Positives

IMR2 to IMR5 encapsulate Probabilistic IRs. Recall that Probabilistic IRs are susceptible to reporting false positives, because they check non-deterministic behaviours. This means that these IMRs can report false positives. The incidence of false positives reported by a testing technique is an important aspect of its effectiveness. We therefore conducted an experiment to determine the false positive rate. We executed a correct version of the system with our test suite 30 times. Encouragingly, only 1/30 false positives were reported by 1 probabilistic IR.

4.3.3 RQ3. What impact does Interlocutory Metamorphic Testing have on the effectiveness of Interlocutory Testing?

In the previous section, we discovered that IMT was effective in the presence of coincidental correctness, and thus successfully extends MT to such scenarios. The union of MT and Interlocutory Testing will also have had an impact on the effectiveness of Interlocutory Testing. This section explores this impact, and therefore addresses RQ3.

The impact that IMT has on the effectiveness of Interlocutory Testing can be ascertained through a comparative study of the techniques. We therefore applied Interlocutory Testing to the same mutants with the same test suite to obtain results that can be compared to those presented above. Interlocutory Testing was performed using 48 IRs (42 Deterministic IRs and 6 Probabilistic IRs), whilst IMT's IMRs only incorporated a subset of these IRs; in particular IMT used 46 IRs — 41 Deterministic IRs and 5 Probabilistic IRs. The remainder of this section details the results of this comparison.

4.3.3.1 Mutation Score and Failure Detection Rate

Table 4.4 shows that Interlocutory Testing (IT) killed 66/80 (32/32 standard and 34/48 coincidentally correct) mutants, and thereby obtained an MS of 82.50%. The difference in performance (as measured by the number of killed and survived classifications) between IT and IMT (which detected 63/80 faults) is not significant (Fisher's Exact Test: $p > 0.05$). The table breaks down the results by IR type. It shows that the effectiveness of the Deterministic IRs varied in IMT and IT. A Fisher's Exact Test was conducted to compare the number of killed and survived classifications proposed by IMT's Deterministic IRs against the number of killed and survived classifications suggested by the Deterministic IRs of IT. The test revealed that the difference was not statistically significant (Fisher's Exact Test: $p > 0.05$). Similar observations were made for Probabilistic IRs.

We calculated the FDR of IMT and IT for all 45 of the mutants that were killed by the deterministic IRs of both techniques. IMT and IT obtained an average FDR of 50.8% and 46.47% respectively. Although the difference was not statistically significant (T-Test[2]: $p > 0.05$), we observed that IMT

[2]The T-Test statistic can be used to compare two groups based on one continuous variable [151]. It is a parametric

obtained a higher FDR for 82.22% of the mutants and IT only obtained a higher FDR for 2.22% of the mutants. We compared the proportion of cases in which IMT had outperformed IT and vice versa, against a hypothetical proportion that represented a scenario in which the two techniques had outperformed each other in the same number of cases, using a Fisher's Exact Test. The difference was statistically significant. This suggests that IMT offers a small improvement in FDR, that can be observed consistently. This level of improvement may be valuable for subtle faults that rarely result in failures.

Given this observation, one would expect Deterministic IRs in IMT to subsume the Deterministic IRs in IT. However, the converse was observed. This means that the higher FDR of IMT did not enable the Deterministic IRs to detect any additional faults, in this experiment. This could be because instances of the aforementioned subtle faults that are detectable by these Deterministic IRs were not present in the experiment. IT outperformed IMT because it used a Deterministic IR, that was not used by IMT, that detected several unique faults. IMT could not make use of this IR because the IR draws on data that is not available in any of the IMR's $MRET$ and $AdditionalET$. This demonstrates that IMT's incapability to leverage certain IRs can hamper its effectiveness.

Figure 4.1: Number of mutants killed by each Probabilistic IR in both techniques

A similar analysis was performed on Probabilistic IRs. Figure 4.1 shows the number of faults that were found by each Probabilistic IR that detected a fault when applied using IT, that were not found by the same IR when applied using IMT, and vice versa. CrossoverRate was used by IT, but not IMT, because this IR requires data that is not available in any of the IMR's $MRET$ and $AdditionalET$. The graph indicates that the Probabilistic IRs performed better in IMT than IT in most cases, in which the IR was available to both techniques. This is likely to be due to the higher FDR in IMT. Interestingly, the converse is also observed in some cases. This indicates that the additional test cases may not have always been effective at exposing failures, because more passes are interpreted by Probabilistic IRs as more evidence that the SUT is behaving correctly. As mentioned above, the difference in performance of Probabilistic IRs in IMT and IT, in terms of MS, was not significant. This could be explained by the demographics of the fault sample used in this experiment; the additional test cases may not have been particularly effective for a large proportion of these faults.

Figures 4.2 and 4.3 collectively present all of the IRs that detected a fault. IRs in Figure 4.2 are associated with a unique IMR and by contrast, Figure 4.3 contains IRs that have a relationship with

test statistic and so is more powerful and sensitive than its non-parametric alternatives [151].

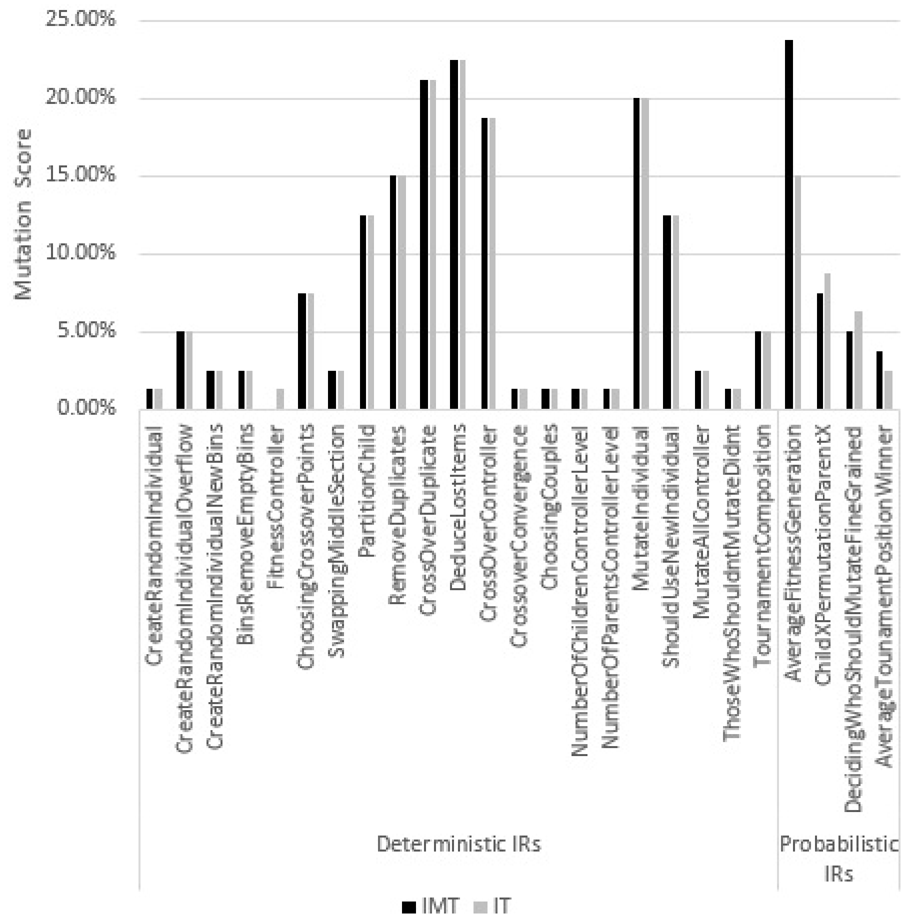

Figure 4.2: IRs that are members of one IMR and their mutation scores

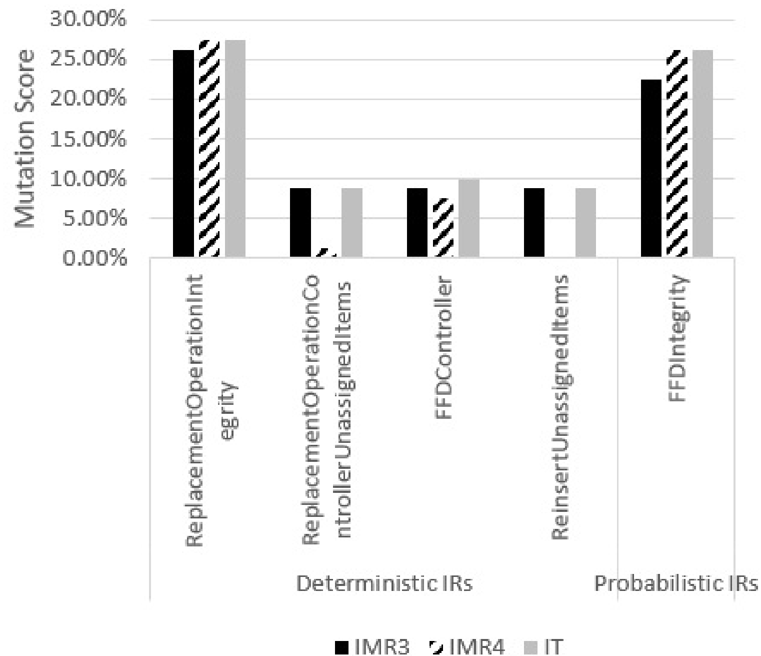

Figure 4.3: IRs that are members of multiple IMRs and their mutation scores

multiple IMRs. Both figures illustrate how their respective IRs performed, when leveraged by IT and IMT.

Interestingly, Figure 4.3 shows that the performance of a Deterministic IR can vary, depending on which IMR it is associated with. A close investigation of this revealed the following: all of the IRs in

Figure 4.3 draw on execution trace data from code that is reused by two distinct genetic operators — crossover and mutation. IMR3 and IMR4 are largely concerned with testing the crossover and mutation operators respectively. This means that although IMR3 and IMR4 share the same subset of source code coverage, they are confined to completely distinct subsets of the execution trace. This clearly means that the MRs in IMR3 and IMR4 expose their IRs to different test data, and indicates that this test data is responsible for the performance of the IRs. This shows that an IMR's coverage in terms of the execution trace as well as source code can be important. This may also explain the performance deviation of the Probabilistic IR, FFDIntegrity, in IMR3 and IMR4.

Let IR be a Deterministic IR that is associated with IMR3 and IMR4. Also let $IMR3_{IR}$ and $IMR4_{IR}$ be the set of faults that were detected by IR, when it was evaluated based on IMR3 and IMR4 respectively. We observed that $IMR3_{IR} \neq IMR4_{IR}$. This suggests that developing multiple IMRs that include the same IRs can add value. The same IR in IMR3 and IMR4 collectively obtained the same MS as the corresponding IR in IT, in all but one case. In this case, the IR had successfully detected a mutant under one technique but not the other, and this mutant was killed by other deterministic IRs, consistently across both techniques. This suggests that different IRs may have different FDRs. It also shows that there is value in including multiple IRs in an IMR that overlap in terms of the faults they can detect.

Interestingly, some IRs that were exposed to test data by IMR3 were more effective than the same IRs in IMR4 and vice versa e.g. ReplacementOperationIntegrity is less effective under IMR3 than IMR4, and ReplacementOperationControllerUnassignedItems is more effective under IMR3. This shows that the test data produced by an IMR can be effective for some IRs, but not others. This suggests that associating an IMR with IRs that will be effective, when these IRs are evaluated based on test data that is produced by this IMR, might be an effective strategy. Unfortunately, the results don't shed light on which groups of IRs would be apposite for certain IMRs, thus future work is necessary to devise some guidelines.

The discussion in this section indicates that IMT and IT may have found different faults. To verify this, we performed an overlap analysis. The analysis revealed that IT found 7 faults that could not be detected by IMT, and that IMT detected 4 faults that IT missed. When combined, the techniques obtain an MS of 87.5%. This suggests that IMT may be a useful complementary technique for IT.

4.3.3.2 False Positive Rates

We measured IT's false positive rate using the same methodology as in Section 4.3.2.3 i.e. a correct version of the system was executed with the test suite 30 times. IT did not report any false positives. The difference between IMT and IT's performance in terms of false positives and true negatives was not statistically significant (Fisher's Exact Test: $p > 0.05$). This means that IMT can offer the improvements discussed above, without substantially reducing the techniques effectiveness in terms of false positives.

4.3.4 RQ4. What effect does the test suite have on the effectiveness of IMT?

This section investigates whether the test suite has a substantial impact on the effectiveness of IMT and thus answers RQ4. This was achieved by using the same test case generation methodology, as

detailed in Section 4.2.3, to generate a second test suite. For ease of reference, we refer to the test suite used in the previous experiments as TS1 and this new test suite as TS2. We executed the first 30 mutants that were generated for the previous experiment with TS2 to obtain TS2's results, and compared these to TS1's results for these mutants. Greater consistency indicates that the test suite has a limited impact, and lower consistency suggests the converse.

IMT, using TS2, killed 22/30 of these mutants, compared to 23/30 in TS1. We compared the performance (as measured by the number of killed and survived classifications) of TS1 and TS2 and found that the difference is not statistically significant (Fisher's Exact Test: $p > 0.05$). This suggests that the test suite had little impact on the additional test cases that were generated by IMT, in terms of their effectiveness for revealing failures. We conducted a similar analysis for the false positive and true negative rates; TS2 reported 0/30 false positives, which is again, not statistically significantly different from TS1, which had only reported 1/30 false positive (Fisher's Exact Test: $p > 0.05$).

4.3.5 Discussion

Let Sys be the SUT. Suppose that one applies MT to Sys, and then applies IT to Sys (this application of IT is independent of the application of MT), or vice versa. Let "MT+IT" denote this scenario. In this section we explore whether IMT is more than just the sum of its constituent parts, by comparing IMT to MT+IT.

Let tc be a test case, and suppose that tc was executed to produce an execution trace LOG. In MT+IT, IRs are evaluated based on LOG. By contrast, in IMT, LOG is first used to generate additional execution trace data (in the form of $AdditionalET$), and IRs are then evaluated based on both LOG and this additional execution trace data. Therefore, IRs in IMT can be evaluated more times within tc than IRs in MT+IT. One consequence of this is that an IR in IMT might be more likely to detect a fault than the same IR in MT+IT. Let PIR be a Probabilistic IR. Another consequence is that the ratio of pass/fail verdicts reported by PIR might be different in MT+IT and IMT. To illustrate, suppose that when evaluated based on execution trace data from LOG with either MT+IT or IMT, PIR passed 9 times and failed once. Further suppose that IMT evaluated PIR 5 additional times based on additional execution trace data, and that all of these evaluations failed. The failure rate of PIR would be 10% in MT+IT, and 40% in IMT. This means that Probabilistic IRs might draw different conclusions i.e. IMT's Probabilistic IRs might be either more or less effective.

Recall that IMT may not be compatible with certain IRs. This means that MT+IT might be able to leverage IRs that cannot be used in IMT; thus MT+IT could be more effective for faults that can be detected by these IRs.

Let MR be an MR that is composed of one source and two follow up test cases. In IMT, an IMR that is associated with MR is responsible for procuring the execution traces of these three test cases ($MRET$, $AdditionalET_1 \in AdditionalET$, and $AdditionalET_2 \in AdditionalET$), and evaluating a set of IRs on these execution traces. One method of implementing this might include adopting an iterative procedure in which one execution trace is procured and used for the evaluation of IRs on each iteration. For example, in this case, the first iteration may procure $MRET$ and evaluate IRs based on $MRET$, the second iteration might procure $AdditionalET_1$ and evaluate IRs based on $AdditionalET_1$, and the last iteration might procure $AdditionalET_2$ and evaluate IRs based on $AdditionalET_2$. If such an implementation approach was adopted, and a Deterministic IR reported a failure during an

early iteration e.g. on the second iteration, then IMT would have sufficient evidence to conclude that the SUT is faulty before all of the execution traces have been procured. One would then have the option to force IMT to terminate prematurely e.g. after the second iteration in this case. Thus, if one were to adopt such an implementation approach, then IMT could be more efficient than MT from MT+IT because it might not be necessary to execute the entire Metamorphic Test Group in IMT. However, as discussed above IMT might involve more IR evaluations than IT from MT+IT and so it could be less efficient than IT. It is unclear to what extent these efficiency deteriorations would be offset by the aforementioned efficiency gains, but there is potential for IMT to have a different level of efficiency, when compared to MT+IT.

The discussions above demonstrate that there is an interplay between MT and IT in IMT, and that this interplay means that IMT is fundamentally different from MT+IT. Therefore, IMT is more than just the sum of its constituent parts.

4.4 Threats to validity

All of the threats to validity that were outlined in Section 3.5 are relevant to the experiments conducted in this chapter. In addition to these threats, we also observed that the experiments only made use of nine IMRs. Many researchers e.g. Liu et al. [109] have reported that a small number of MRs is sufficient. We therefore believe that our sample size of IMRs was representative. Focusing on a small number of IMRs also afforded us the opportunity to do some detailed analysis that would not have been possible with a larger sample size.

One of the threats to validity in Section 3.5 was concerned with the mutant sample size. We would like to remark that the results revolving around the overall effectiveness of Interlocutory Testing in this chapter was comparable to the corresponding results in Chapter 3. This stability in the results suggests that the mutant sample was large enough to characterise the typical distribution of faults that can and can't be detected by our IRs.

The implementation of AverageFitnessGeneration was slightly different in IT and IMT. Both of these implementations are viable alternatives i.e. the difference does not represent a fault. Thus, the results pertaining to each individual technique has not been adversely affected. However, there is potential for this difference to confound the comparisons of these techniques. We investigated this possibility and found that had IT used the same implementation of this IR as IMT, the results would have been exactly the same. Alternatively, had IMT used the same implementation of the IR as IT, the IR would have killed one less mutant. This mutant was killed by another probabilistic IR. Thus, this difference either has no impact, or a negligible impact that does not have any meaningful consequences for the results, depending on whether the former or latter implementation is adopted respectively.

Due to a version control error, an older version of the FitnessController IR was used by IMT. We investigated the impact that this might have on the results. We found that this IR would have detected one additional mutant, had the new version been used. We also observed that other IRs successfully detected this mutant, and obtained an FDR of 100% for this mutant. Thus, this error has no consequences for the overall results, and only minor, immaterial implications for the lower level results.

In our implementation of IMT, some of our IMT code reuses the system's source code (e.g. to generate a follow-up test case for an IMR in our Genetic Algorithm subject program, the IMR might use the SUT's individual factory class), and thus extends the coverage of our test cases. One threat to repeatability might be that other's adopt an alternative implementation approach, in which the system's source code is not reused.

Finally, it might have been possible for certain mutants to cause the IMT code (e.g. code that logs execution trace data that is necessary for MR evaluation) to crash during the execution of these mutants. In such situations, IMT has effectively detected the mutant, but we did not distinguish between such crashes and system crashes, and so would have conservatively removed such mutants. This means that IMT's effectiveness may have been underestimated. We note however, that IMT detected most of the mutants, and so this is unlikely to have had a significant impact on the results.

4.5 Conclusions

In this chapter, further evidence that suggests that MT is ineffective in the presence of coincidental correctness was ascertained. We also introduced IMT, an extended version of MT that can conduct testing in the presence of coincidental correctness. The feasibility of IMT was demonstrated on four subject programs and a sample of 40 mutants. IMT was also shown to be effective in testing scenarios that involve coincidental correctness and the oracle problem — IMT obtained an MS of 78.75%, an FDR of 50.8% and only reported 1/30 false positives. Thus, we illustrated that IMT can extend the generalisability of MT to such scenarios. These findings suggest that Objective 2 has been satisfied (see Section 1.1).

We also performed a comparative analysis between IMT and IT to determine what the impact of IMT is on IT. We observed that the techniques can find different faults. IT can find faults that cannot be detected by IMT because it may have access to IRs that are not available to IMT, and IMT can find additional faults because it has a higher FDR. This suggests that IMT would be a useful complementary technique for IT.

Various insights into the effectiveness of IMT were also obtained. For example, we observed that some IMRs empowered some IRs, whilst others decreased the effectiveness of some IRs. Such insights have revealed promising future research directions. For example, it's unclear which IRs should be associated with which IMRs to enhance their effectiveness; future research that explores this would be beneficial. Our comparative analysis between IMT and IT provided some evidence that suggested that IMT had various effects on IT. For example, since IMT has a higher FDR, it's more likely to find a subtle fault than IT. However, we did not observe any such cases for the Deterministic IRs, and a non-significant number of cases for Probabilistic IRs. This is likely to be because there was not a substantial number of appropriate faults in the experiment to demonstrate the effect more significantly. Thus, future work that explores this further in different testing scenarios may be beneficial.

To summarise, this chapter modified Metamorphic Testing, by integrating Interlocutory Testing into it. The result of this integration was a reduction in the susceptibility of Metamorphic Testing to coincidental correctness. Thus, this chapter fulfilled Objective 2 of the thesis. In the next chapter, we explore whether Interlocutory Testing can be used as a partial solution to the Equivalent Mutant Problem, that can tolerate coincidental correctness and non-determinism.

Chapter 5

Interlocutory Mutation Testing

Techniques that are used to classify mutants as either equivalent or non-equivalent, and thereby resolve the Equivalent Mutant Problem, can be inaccurate for systems that are susceptible to coincidental correctness and/or that are non-deterministic (see Section 1.1.3). Despite the prevalence of coincidental correctness, very few solutions have been proposed to ameliorate the effects of coincidental correctness on such techniques (see Section 2.8.2). Similarly, relatively few solutions have been proposed for non-deterministic systems (see Section 5.4). This motivated the work that is presented in this chapter. In Chapter 3, we introduced a new testing technique called Interlocutory Testing, that can suppress the effects of coincidental correctness, and operate effectively in the presence of non-determinism. This chapter explores how Interlocutory Testing can be used to alleviate the Equivalent Mutant Problem in such systems. We call the approach Interlocutory Mutation Testing (IMuT). Thus, this chapter attempts to address Objective 3 (see Section 1.1).

This chapter makes the following main contributions:

1. A new technique called IMuT that can classify mutants as equivalent or non-equivalent in programs with coincidental correctness and/or non-deterministic behaviours.
2. An evaluation of the accuracy of IMuT, and an assessment of whether the results obtained from this evaluation can generalise.
3. An evaluation of the impact IMuT might have on productivity, and an assessment of whether the results obtained from this evaluation can generalise.
4. An experiment that compares IMuT to TEMDT.

IMuT is introduced in Section 5.1. Section 5.2 describes the experimental design for a series of experiments, which are presented in Section 5.3. In Section 5.4, we present related work revolving around the Equivalent Mutant Problem and non-deterministic systems, and Weak Mutation Testing. Related work on the Equivalent Mutant problem in the context of coincidental correctness can be found in Section 2.8.2. Other related work is discussed in Section 3.4. Threats to validity are outlined in Section 5.5, and finally, conclusions are drawn in Section 5.6.

5.1 Interlocutory Mutation Testing — Technique Description

Algorithm 6: Bubble Sort

```
   Input: A sequence of integers List
   Output: A modified version of List
 1 for j = 1 to n, where n = List.size() do
 2     for i = 0 to n − 2 do
 3         if List.get(i) > List.get(i + 1) then
 4             List = swap(List, i, i + 1);
 5         end
 6     end
 7 end
 8 //List.set(0, randomInteger());
```

IMuT was developed to enable the classification of equivalent and non-equivalent mutants for programs that are non-deterministic and/or are susceptible to coincidental correctness. This section introduces IMuT and uses two versions of the Bubble Sort algorithm as a running example. Algorithm 6 describes one of these versions; this version is the original version and will henceforth be referred to as the $Bubble_o$. The other version, which will be referred to as $Bubble_m$, is a non-equivalent mutant of $Bubble_o$, in which line 8 is uncommented. In $Bubble_m$, the value of the first element of the output is overwritten with a random value. For more complicated examples that include non-determinism and coincidental correctness, please see Section 3.1.

5.1.1 Intuition

Interlocutory Testing is an effective means of conducting testing, despite the presence of coincidental correctness and non-determinism. To briefly recap; let S denote the SUT. In Interlocutory Testing, an IR predicts a set of execution trace behaviours, such that these execution trace behaviours are expected to manifest in S. These predictions are verified by checking whether they actually manifest in S.

Let M be a mutant of S. Interlocutory Testing can be modified to support the mutant classification process as follows. Instead of being designed to predict behaviours that are expected to manifest in S, IRs are designed to predict behaviours that actually manifest in S, and instead of verifying these predictions by checking that they actually manifest in S, these predictions are verified by checking that they actually manifest in M. This modified version of Interlocutory Testing is tantamount to checking whether the behaviours of M are the same as the behaviours of S. This is the intuition behind IMuT.

5.1.2 Technique Description

Algorithm 7: Interlocutory Mutation Testing

Input: Let ts be a test suite consisting of n test cases. Suppose that each test case $tc_i \in ts$ was executed on a mutant M of the original version of the SUT S. Let $LOGs = \{LOG_1, LOG_2, ..., LOG_n\}$ be the resultant set of execution trace log files. $LOGs$ is the input for this algorithm.

Output: $Classification$

1 Let IRs be a set of IRs, such that each $IR \in IRs$ makes predictions that are based on how S actually operates;

2 IRs in IRs are evaluated based on $LOGs$, using the same methodologies that were described in Section 3.1. Let $Results$ denote the outcomes of these evaluations;

3 **if** *At least one of the outcomes in Results is SUTFaulty* **then**

4 $\quad Classification$ = Non-equivalent Mutant;

5 **else**

6 $\quad Classification$ = Equivalent Mutant;

7 **end**

Algorithm 7 describes the process of applying IMuT. This section explains the process in detail.

Let S be a system e.g. $Bubble_o$. In IMuT, IRs are developed (see Section 3.1) based on S. For example, let *Input* and *Output* be two sequences of integers. *Input* and *Output* are the input and output of the Bubble Sort algorithm respectively. An IR called BubbleIR may be developed, and be associated with the following IOR: $Input \neq Output$, and this IOR may be associated with an ID that predicts that the Swap Operator was invoked at least once. In Interlocutory Testing, IRs are designed to make predictions that are based on the tester's intentions for how S should operate. By contrast, IRs in IMuT are designed to make predictions that are based on how S actually operates. Line 1 of Algorithm 7 defines IRs to be a set of such IRs.

As an aside, this means that unlike IRs that are based on Interlocutory Testing, IRs that are based on IMuT cannot detect faults in S. It's also worth mentioning that there can be some degree of overlap between the IRs in Interlocutory Testing and IMuT in practice, because IRs can be designed to make predictions that are based on the tester's intentions for how the S should operate, and also accurately reflect how S actually operates.

Let M be a mutant that was derived from of S e.g. $Bubble_m$. The input for Algorithm 7 is a set of execution traces that were produced by test cases that were executed on M. Line 2 of Algorithm 7 evaluates the IRs in IRs against these execution traces (using the same methodologies that were described in Section 3.1). To illustrate, let $MInput$ and $MOutput$ denote the input and output of $Bubble_m$ respectively. The mutation in $Bubble_m$ can lead to situations in which $MInput \neq MOutput$ and the swap operator was not invoked; thus BubbleIR's prediction could be incorrect. Lines $3-7$ of Algorithm 7 leverage the outcomes of these evaluations to classify the mutant. In particular, if at least one of the outcomes is SUTFaulty, then Algorithm 7 concludes that M is an non-equivalent mutant, because this indicates that the behaviour of M deviated from the behaviour of S. Conversely, if all of the outcomes are SUTPossiblyCorrect, then this indicates that the behaviour of M did not deviate from the behaviour of S, and so M is classified as an equivalent mutant. In the case of our example, BubbleIR's prediction could be wrong; thus $Bubble_m$ can be classified as a non-equivalent mutant.

As discussed above, IMuT assumes that an IR is encoded with accurate information about how S works. It is important to note that, this assumption may not hold if a real fault exists in the system or IRs. To reduce the impact of this assumption, we recommend applying the IRs to S with a test suite. If any of the IRs indicate that the S is non-equivalent, then the assumption doesn't hold. In such cases, one can either modify the system and/or IRs, or remove IRs until all IRs report that S is equivalent. The same test suite should then be used for conducting IMuT.

5.2 Experimental Design

This section presents our research questions and the design of the experiments that were conducted to explore them.

5.2.1 Research Questions

RQ1: How accurate is IMuT in the presence of coincidental correctness and non-determinism? The primary goal of IMuT is to be an effective means of classifying mutants as either equivalent or non-equivalent, in the presence of coincidental correctness and/or non-determinism. This research question explores its aptitude for this task.

RQ2: What impact might IMuT have on productivity? One motivation for the use of automated mutant classification techniques is to improve one's productivity. This research question attempts to quantify the productivity gains that one can obtain by using IMuT.

RQ3: How does IMuT compare to other mutant classification techniques? This research question attempts to quantify the difference in the effectiveness of our technique against the effectiveness of widely used mutant classification techniques.

RQ4: Can our findings revolving around the accuracy of IMuT generalise to deterministic systems without coincidental correctness? This research question explores whether our results pertaining to the accuracy of IMuT can generalise to other testing contexts.

RQ5: Can our findings revolving around the productivity gains offered by IMuT generalise to other problem domains? This research question explores the generalisability of our results pertaining to the productivity gains offered by IMuT.

We conducted a series of experiments to answer the research questions above. For ease of reference, we call the three experiments that were conducted to address RQ1 — RQ3 the main experiments, the experiment that explores RQ4 the Generalise-Accuracy experiment, and the experiment that investigates RQ5 the Generalise-Productivity experiment.

5.2.2 Subject Programs

A total of five subject programs are used across the experiments. One subject program is used in the main experiments; a Genetic Algorithm for the Bin Packing Problem — a description of this subject program can be found in Section 3.2.1.1. This subject program was selected for the same reasons that were discussed in Section 3.2.1.1. Similarly, the Generalise-Accuracy experiment also uses one subject program — an implementation of Dijkstra's Algorithm. Details about this subject

program are presented in Section 3.2.3.1. This subject program was selected for this experiment because it is deterministic and is unlikely to be susceptible to coincidental correctness, since its information flows aren't particularly weak e.g. there are very few overwriting operations that could obscure enough information to mask a fault. This makes it ideal for answering RQ4. Finally, the Generalise-Productivity experiment uses three subject programs: Bubble Sort, Binary Search and Knuth-Morris-Pratt. Please see Section 3.2.3.1 for a description of these subject programs. Our reasons for selecting these subject programs can be found in Section 3.2.3.1.

5.2.3 Interlocutory Relations

A total of 57 IRs were used across all of the experiments. A list of the 48, 4, 1, 1, and 3 IRs that were used for Genetic Algorithm, Dijkstra's Algorithm, Bubble Sort, Binary Search, and Knuth-Morris-Pratt subject programs respectively, can be found in Appendices A, C, D, E and F respectively. Each IR is also associated with a summary of the main aspects of that IR. These groups of IRs were deemed to be large enough to support the types of analysis that were conducted based on these groups.

5.2.4 Mutants

Refactoring is a technique that changes the internal structure of source code, without changing its observable behaviour [63]; this is essentially what equivalent mutants are. Thus, automated refactoring tools can be used to generate equivalent mutants. AutoRefactor is such a tool [164]. We used AutoRefactor to generate most of the equivalent mutants because all mutants produced by AutoRefactor were guaranteed to be equivalent mutants. This eliminated the need for manual inspection (for some of the mutants) that would have otherwise been necessary had MuJava been used. Thus, the use of AutoRefactor reduced the potential for human error. IMuT was exposed to total of 113 non-equivalent mutants and 11 equivalent mutants that were generated by MuJava (see Section 3.2.1.2), and 60 equivalent mutants that were generated based on refactorings produced by AutoRefactor, across all of the experiments.

5.2.4.1 Main Experiments

The first of the main experiments was designed to address RQ1. This experiment uses two sets of mutants: 30 non-equivalent mutants and 30 equivalent mutants. We obtained the set of 30 non-equivalent mutants, by using the same mutant generation strategy that was outlined in Section 3.2.1.2. 15 of these mutants were coincidentally correct, and 15 were standard faults. Our justifications for the use of this strategy are presented in Section 3.2.1.2. We used AutoRefactor to generate a set of 30 random equivalent mutants. Two samples of 30 equivalent and 30 non-equivalent mutants were large enough to support an investigation into the mutant classification accuracy of IMuT for equivalent and non-equivalent mutants respectively.

The second of the main experiments was developed to explore RQ2. We conjectured that one's mutant classification productivity might be affected by the proportion of the mutant sample that was equivalent and non-equivalent. To that end, we felt that it was necessary to generate a mutant sample that contained both types of mutants. We therefore generated a sample of 30 mutants; the sample consists of 7 equivalent and 23 non-equivalent mutants. To obtain these mutants, we used a modified version of the mutant generation strategy that is outlined in Section 3.2.1.2; the difference

between the two strategies, is that the mutant generation strategy used here does not reject equivalent mutants. All of the justifications presented in Section 3.2.1.2, apart from the justification concerning the rejection of equivalent mutants, are relevant for this experiment.

The last of the main experiments evaluates RQ3. Only 30 equivalent mutants were required for this experiment. We decided to leverage the same equivalent mutants that were used in the first of the main experiments in this experiment, because using the same mutant sample would increase our confidence in the results of comparative analyses that are based on the results of these experiments. 30 equivalent mutants were sufficient to demonstrate that TEMDT could not provide any mutant classification support in situations in which coincidental correctness and non-determinism are present.

5.2.4.2 Generalise-Accuracy Experiment

This experiment leverages 30 non-equivalent and 30 equivalent mutants. These 30 non-equivalent mutants were generated using the same strategy that was presented in Section 3.2.3.2. We applied AutoRefactor to the subject program and found that it could not generate a sufficient number of equivalent mutants. To that end, we retained all of the equivalent mutants that were generated by the tool. Let $AllMutants$ denote these mutants. The following procedure can be used to generate an equivalent mutant, based on $AllMutants$: select a random subset of mutants from $AllMutants$, and then create a new mutant, such that this mutant contains all of the mutations from the random subset of selected mutants. This procedure was used multiple times to generate additional equivalent mutants, to supplement $AllMutants$ and make up the deficit. Our justifications with regards to the mutant generation strategy that was used for the non-equivalent mutant sample are the same as those that were presented in Section 3.2.3.2, and our reasons concerning the use of one non-equivalent and one equivalent mutant sample, consisting of 30 mutants each, are the same as those that were given for the first main experiment (see Section 5.2.4.1).

5.2.4.3 Generalise-Productivity Experiment

This experiment used a total of 30 non-equivalent mutants and 4 equivalent mutants, across the Bubble Sort, Binary Search, and Knuth-Morris-Pratt subject programs. 10 non-equivalent mutants were generated for each of these subject programs, using the procedure outlined in Section 3.2.3.2. All of the equivalent mutants that were generated during the production of these 30 non-equivalent mutants were retained, to be used as the set of equivalent mutants in this experiment; 0, 2, and 2 equivalent mutants were produced for the Bubble Sort, Binary Search, and Knuth-Morris-Pratt subject programs respectively. All of the justifications that were posited in Section 3.2.3.2 are relevant for this experiment, apart from the justification regarding the rejection of equivalent mutants. The decision to use a single sample of mutants that consists of both equivalent and non-equivalent mutants was motivated by the same reasons that were discussed for the second main experiment (see Section 5.2.4.1).

5.2.5 Test Cases

100 test cases were procured for each subject program; thus there were a total of 500 test cases across all of the experiments. The 100 test cases that were used for the Genetic Algorithm subject program, were the same test cases that were used in Chapter 3. The test cases for the other subject programs were generated using the same test case generation strategies that were outlined in Section 3.2.3.3.

The reasoning behind the use of these test case generation strategies and the test suite from Chapter 3 is presented in Sections 3.2.3.3 and 3.2.1.3 respectively.

5.2.6 Benchmark

As discussed in Section 1.1.3, TEMDT is one of the most widely used methods for estimating whether a mutant is equivalent or not. We therefore selected it as a benchmark technique because many researchers will be familiar with such a technique's level of effectiveness.

5.3 Results and Discussion

5.3.1 RQ1. How Accurate is IMuT in the Presence of Coincidental Correctness and Non-determinism?

IMuT's accuracy can be determined by analysing the proportion of equivalent and non-equivalent mutants that were correctly and incorrectly classified. To conduct such an analysis, we leveraged the Genetic Algorithm subject program, 30 non-equivalent mutants (generated by MuJava), 30 equivalent mutants (generated by AutoRefactor), and 100 test cases. Sections 5.3.1.1 and 5.3.1.2 presents our results for non-equivalent and equivalent mutants respectively.

5.3.1.1 Non-Equivalent Mutants

IMuT correctly classified 28/30 non-equivalent mutants. This suggests that IMuT's classification accuracy can be high for non-equivalent mutants. Since the SUT is non-deterministic, this also demonstrates that the technique's classification accuracy for these mutants was not hampered by non-determinism. Specifically, 15/15 and 13/15 standard and coincidentally correct mutants were correctly classified. We compared the number of correct and incorrect classifications made by IMuT for standard mutants against such classifications for coincidentally correct mutants; the difference is not significant (Fisher's Exact Test: $p > 0.05$). This indicates that IMuT can be effective for standard and coincidentally correct faults.

Recall that there are two types of IRs — Deterministic and Probabilistic IRs. Also recall that we have a sample of 42 Deterministic IRs and 6 Probabilistic IRs. These IRs are distinguished by the types of logic they are applied to — deterministic IRs are applied to aspects of the system that behave deterministically, whilst probabilistic IRs are applied to non-deterministic aspects of the system. To that end, each approach has different evaluation methods; the difference being, Probabilistic IRs leverage statistical techniques to factor out the effect of false positives that arise due to non-determinism. We therefore decided to further break down the analysis by these IR types.

Deterministic IRs correctly classified 23/30 (13/15 standard and 10/15 coincidentally correct) non-equivalent mutants. The number of correct and incorrect classifications made by Deterministic IRs for standard mutants, was not significantly different to the number of such classifications made by these IRs for coincidentally correct mutants (Fisher's Exact Test: $p > 0.05$). This demonstrates that one can leverage these IRs in contexts where coincidental correctness is present, or absent. Each bar in Figure 5.1 represents a Deterministic IR that correctly classified a mutant. The height of the bar denotes the number of correctly classified non-equivalent mutants. Each bar also represents the

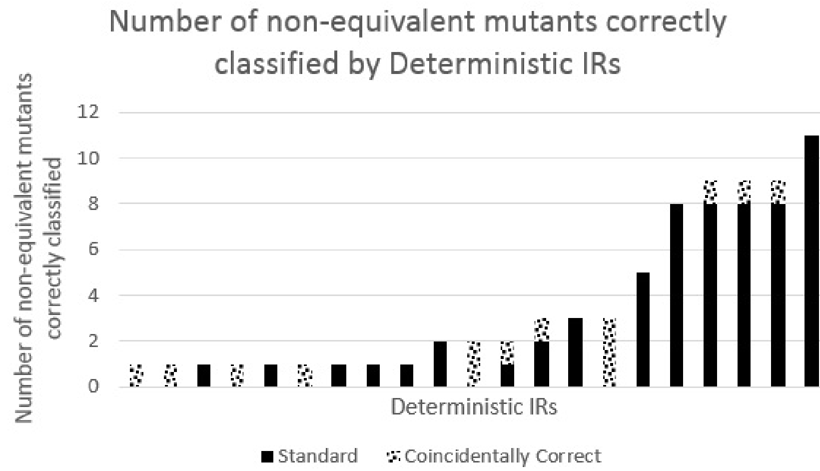

Figure 5.1: Number of mutants that were correctly classified by Deterministic IRs, broken down by mutant type

proportion of mutants that were standard or coincidentally correct. Figure 5.1 demonstrates that some IRs are more accurate than others for different mutants. For example, the IR represented by the third bar correctly classifies standard mutants, but not coincidentally correct mutants, and the converse is true for the IR that is represented by the second bar.

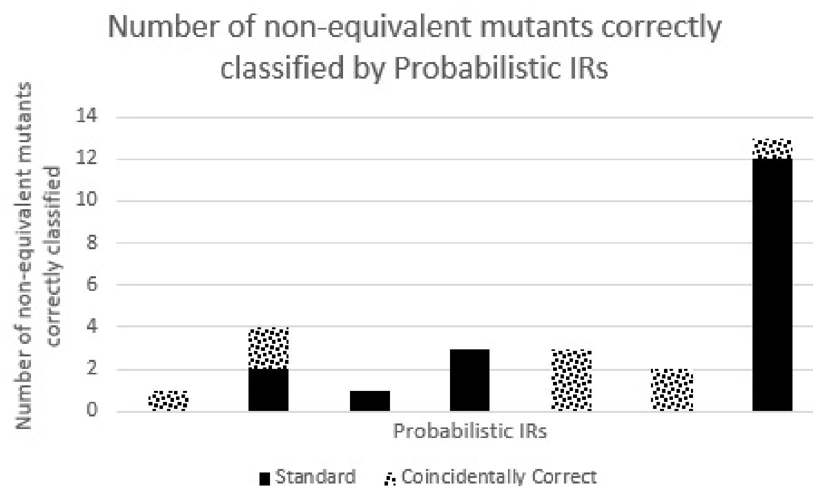

Figure 5.2: Number of mutants that were correctly classified by Probabilistic IRs, broken down by mutant type

21/30 (14/15 standard and 7/15 coincidentally correct) non-equivalent mutants were correctly classified by Probabilistic IRs. A comparison of the performance (as measured by the number of correct and incorrect classifications) of Deterministic and Probabilistic IRs for standard faults revealed that the difference was not statistically significant (Fisher's Exact Test: $p > 0.05$). Similarly, the difference was not statistically significant for coincidentally correct faults (Fisher's Exact Test: $p > 0.05$). This suggests that the effectiveness of Probabilistic IRs is comparable to Deterministic IRs in situations in which coincidental correctness is present or absent. Since Probabilistic IRs have the potential for reporting false positives that arise due to non-determinism, this indicates that one should prioritise Deterministic IRs over Probabilistic IRs, since IMuT would achieve a similar level of effectiveness, without introducing the risk of reporting such false positives. However, we observed that 3 of the coincidentally correct faults, and 2 of the standard faults that were found by IMuT were uniquely

identified by Probabilistic IRs, which means that they can add value. Figure 5.2 presents the same information as in Figure 5.1, but for Probabilistic IRs; similar observations can be made to those in Figure 5.1.

As discussed above, all of the IRs collectively, correctly classified 28/30 non-equivalent mutants. Deterministic IRs and Probabilistic IRs correctly classified 23 and 21 mutants respectively, which means that neither IR type correctly classified all of the mutants on their own. This demonstrates that both IR types can add value.

Interestingly, these results also suggest that there was a substantial degree of overlap in terms of the number of mutants that were correctly classified by the IRs. We therefore decided to perform a subsumption analysis to determine the smallest number of IRs that would be required to obtain the same results. We found that only 10 were necessary: AverageFitnessGeneration, CreateRandomIndividualNewBins, CrossoverRate, DecidingWhoShouldMutateFineGrained, DeduceLostItems, FFDIntegrity, GAController, MutateIndividual, ShouldUseNewIndividual, and TournamentComposition. This shows that the technique can be effective with relatively few IRs.

Let $IMuTSubsumption$ denote results of the subsumption analysis above (i.e. the 10 IRs listed above), and $IMuTMutants$ be the mutant sample that was used in this subsumption analysis. Recall that we performed another subsumption analysis in Section 3.3.2.2; similarly, let $ITSubsumption$ denote the results of that subsumption analysis (14 IRs), and $ITMutants$ be the mutant sample that was used in that subsumption analysis. We decided to perform a comparative analysis between $IMuTSubsumption$ and $ITSubsumption$. We observed that the sizes of $IMuTSubsumption$ and $ITSubsumption$ were different. We also found that, even though the majority of IRs in $IMuTSubsumption$ were also in $ITSubsumption$ and vice versa ($IMuTSubsumption$ overlapped with $ITSubsumption$ on 8 IRs), $IMuTSubsumption$ did not overlap with $ITSubsumption$ on all of the IRs. In particular, 6 of the IRs in $ITSubsumption$ did not appear in $IMuTSubsumption$ and 2 of the IRs in $IMuTSubsumption$ did not appear in $ITSubsumption$. This demonstrates that the most effective, smallest set of IRs was different for $IMuTMutants$ and $ITMutants$.

The main differences between the experiments that were conducted to obtain $IMuTSubsumption$ and $ITSubsumption$ were the composition of the mutant samples and non-determinism. Thus, these may explain some of the differences between $IMuTSubsumption$ and $ITSubsumption$. Let IR_a and IR_b be two IRs, such that IR_a finds exactly the same faults as IR_b. In such cases, the choice of retaining either IR_a or IR_b is arbitrary. Thus, this may also partly explain some of the differences in $IMuTSubsumption$ and $ITSubsumption$.

Recall that $ITSubsumption$ correctly classified 87/100 mutants, when it was applied to $ITMutants$. We decided to apply $ITSubsumption$ to $IMuTMutants$ to determine its accuracy for these mutants; we found that it could correctly classify 26/30 mutants. A comparison of the effectiveness (as measured by the number of correct and incorrect classifications) of the application of $ITSubsumption$ to $ITMutants$ against the application of $ITSubsumption$ to $IMuTMutants$ revealed that the difference was not statistically significant (Fisher's Exact Test: $p > 0.05$). This suggests that $ITSubsumption$ can obtain a comparable level of effectiveness for different mutant samples. Also recall that $IMuTSubsumption$ correctly classified 28/30 of the mutants in $IMuTMutants$. We applied $IMuTSubsumption$ to $ITMutants$; 70/100 mutants were correctly classified. A comparison of the effectiveness (again, in terms of the number of correct and incorrect classifications) of $IMuTSubsumption$ for these mutant

samples yielded statistically significant results (Fisher's Exact Test: $p < 0.05$). The results of this comparison indicate that, unlike *ITSubsumption*, the effectiveness of *IMuTSubsumption* is partly dictated by the mutant sample.

We decided to compare the effectiveness of *ITSubsumption* and *IMuTSubsumption* on the different mutant samples. We found that the effectiveness of *ITSubsumption* on *ITMutants* (87/100) was substantially different from *IMuTSubsumption* on *ITMutants* (70/100), and that the effectiveness of *ITSubsumption* on *IMuTMutants* (26/30) was comparable to *IMuTSubsumption* on *IMuTMutants* (28/30). These results suggest that *ITSubsumption* is preferable to *IMuTSubsumption* because it outperformed *IMuTSubsumption* in the best case, and obtained comparable results in the worst case. However, we observed that *ITSubsumption* consists of 4 more IRs than *IMuTSubsumption*, and so is more expensive. This means that *IMuTSubsumption* may be more preferable in situations in which its effectiveness is comparable to *ITSubsumption*.

5.3.1.2 Equivalent Mutants

Promisingly, IMuT correctly classified 30/30 equivalent mutants. The classification accuracy of the Deterministic IRs for these equivalent mutants may partly be explained by the fact that Deterministic IRs don't check non-deterministic aspects of the system, and so are not susceptible to false positives that arise from non-determinism, and because the assumption detailed in Section 5.1 held.

In theory, a Deterministic IR could report a false positive, if an equivalent mutant's mutation prevents certain execution trace behaviours from manifesting. To illustrate, let $F_1()$ and $F_2()$ be two functions that implement the same functionality. An equivalent mutant might invoke $F_1()$ in the place of $F_2()$, and a Deterministic IR that expects $F_2()$ to have been invoked might resultantly conclude that the mutant is non-equivalent. The results indicate that such scenarios are unlikely.

Since Probabilistic IRs check non-deterministic aspects of the system, false positives that arise from non-determinism may be possible. To that end, we extended the evaluation method used by Probabilistic IRs, as described in Section 3.1.2, to curtail the incidence of false positives. These results illustrate that this evaluation method was successful in achieving this goal.

5.3.2 RQ2: What Impact Might IMuT have on Productivity?

5.3.2.1 Impact on Manual Inspection Effort

Deterministic IRs are very unlikely produce false positives, as long as the assumption detailed in Section 5.1 holds. This is supported by our findings on the accuracy of Deterministic IRs for equivalent mutants; they obtained 100% classification accuracy (see Section 5.3.1.2). This means that all non-equivalent mutant classifications suggested by Deterministic IRs can be trusted, since it is likely that none of the equivalent mutants would have been misclassified as non-equivalent. Thus, the manual effort required to inspect mutants for equivalence can be reduced because testers don't have to check any of the mutants that have been classified as non-equivalent by Deterministic IRs.

However, the results also demonstrated that Deterministic IRs can misclassify non-equivalent mutants as equivalent. Thus, one's confidence in their equivalent mutant classifications will be lower and so it will be necessary for testers to manually inspect mutants that have been classified as equivalent by these IRs.

Similarly, the results indicate that Probabilistic IRs can misclassify non-equivalent mutants as equivalent; thus manual inspection of these classifications is necessary. Although Probabilistic IRs can theoretically report false positives and thus misclassify equivalent mutants as non-equivalent, the results suggest that this scenario is improbable. Thus, one can have a high degree of confidence in non-equivalent mutant classifications from Probabilistic IRs, and so might choose not to manually inspect mutants that are marked as non-equivalent by these IRs. Again, this can lead to a reduction in manual inspection effort.

In summary, one can trust non-equivalent mutant classifications, but not equivalent mutant classifications. To empirically investigate how this might affect productivity, we used MuJava to generate 30 random mutants based on the Genetic Algorithm subject program; see Section 5.2.4.1. This random sample contained 7 equivalent and 23 non-equivalent mutants.

IMuT was applied to these mutants with 100 test cases; it correctly classified 24/30 mutants. 6/7 equivalent mutant classifications, and 18/23 non-equivalent mutant classifications were correct. Since one can trust all of the non-equivalent mutant classifications, but not the equivalent mutant classifications, this means only 11/30 (i.e. the 6 correctly classified equivalent mutants, and 5 non-equivalent mutants that were misclassified as equivalent) mutants must be manually inspected. This reduces the total number of mutants that must be manually inspected by 63.33%. These reductions are substantial.

5.3.3 RQ3: How does IMuT Compare to Other Mutant Classification Techniques?

We used the same subject program, test suite, and 30 equivalent mutants that were used to explore RQ1 as the basis for a comparison of IMuT and TEMDT. Let $TS = \{tc_1, tc_2, ..., tc_{100}\}$ be a sequence of test cases, and denote the test suite. We first executed the subject program Sys with each test case in TS to obtain a corresponding sequence of outputs $Outp = \{o_1, o_2, ..., o_{100}\}$, where o_i represents the output of tc_i. To apply TEMDT to one equivalent mutant we applied the following procedure: we executed the equivalent mutant $FSys$ with TS to obtain another corresponding sequence of outputs $FOutp = \{fo_1, fo_2, ..., fo_{100}\}$, where fo_i represents the output of tc_i. For each test case $tc_i \in TS$, the corresponding outputs $o_i \in Outp$ and $fo_i \in FOutp$ are compared. A single comparison is achieved as follows. The subject program has two elements in its output; the solution and fitness value. The solution and fitness value in o_i is compared to the solution and fitness value in fo_i respectively. If either of these comparisons reveal a discrepancy, then the equivalent mutant is said to have been misclassified as non-equivalent. This procedure was repeated once, for each equivalent mutant.

The results indicated that TEMDT could not classify any of the equivalent mutants correctly. We believe that this is due to non-determinism. This suggests that the technique can't distinguish between equivalent and non-equivalent mutants in this scenario. This substantially contrasts with IMuT and thus demonstrates the effectiveness of our approach.

Recall that TEMDT's classification of equivalent mutants in this experiment was based on two different aspect of the output i.e. the solution s and fitness value f. We observed that had the experiment only been conducted based on s, or only conducted based on f, none of the mutants would have been classified correctly. However, we also observed that the extent to which each mutant would have been misclassified would have been substantially different. In particular, in the situation

in which only s was used, the average number of test cases that misclassified an equivalent mutant would have been 56.83/100 test cases, and in a scenario in which only f was used, the average number of test cases that did not classify the equivalent mutant correctly would have been 23.50/100. We believe this is due to output cardinality i.e. they have a different number of possible outputs for the same input. There are an enormous number of item and bin permutations which means s has high output cardinality. By contrast, many of these permutations score the same fitness value, thus f has a lower output cardinality than s. Lower output cardinalities means there is less opportunity for the outputs between two executions to deviate. This suggests that using different subsets of the output for TEMDT can have a substantial impact on its effectiveness. We believe that research that explores which subsets are particularly effective will therefore be beneficial.

5.3.4 RQ4: Can Our Findings Revolving Around the Accuracy of IMuT Generalise to Deterministic Systems without Coincidental Correctness?

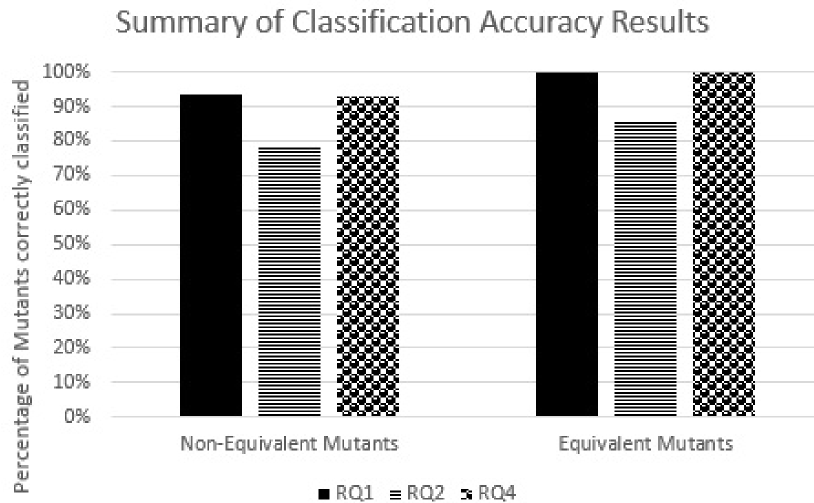

Figure 5.3: Summary of results

To assess RQ4, we repeated the experiment used to evaluate RQ1, on a deterministic system that is less susceptible to coincidental correctness — Dijkstra's Algorithm. 30 non-equivalent mutants, 30 equivalent mutants, and 100 test cases were also used in this experiment. Figure 5.3 is a bar chart that summarises the results of this and the previous experiments conducted in this chapter. There are six bars which are divided into two clusters; one cluster contains the results for non-equivalent mutants, and the other consists of the results for equivalent mutants. Each cluster has three bars, each of which represents one of the experiments undertaken in this chapter. RQ1, RQ2, and RQ4 refer to the experiments described in Sections 5.3.1, 5.3.2, and in this section respectively. The Y-Axis depicts classification accuracy.

Figure 5.3 shows that RQ1, RQ2, and RQ4 achieve an equivalent mutant classification accuracy of 100%, 85.71%, and 100% respectively. There is no difference between RQ1 and RQ4, and the differences (in terms of the number of correct and incorrect classifications) between these experiments and RQ2 is not statistically significant (Fisher's Exact Test: $p > 0.05$). Similarly, no difference was observed between RQ1 and RQ4 with regards to non-equivalent mutant classification accuracy, and

the difference (again, in terms of the number of correct and incorrect classifications) between RQ2 and RQ4 was not found to be statistically significant (Fisher's Exact Test: $p > 0.05$). This means that IMuT's accuracy was consistent across both subject programs. This suggests that many of our findings about IMuT's mutant classification accuracy may generalise to broader types of systems.

We also made several other noteworthy observations regarding IMuT's effectiveness based on Dijkstra's algorithm. For example, only Deterministic IRs were used in this experiment because we couldn't identify any Probabilistic IRs. This demonstrates that certain IR types may not be available in all systems. Since Probabilistic IRs were shown to be effective in the experiments addressing RQ1 and RQ2, this could mean that the effectiveness of IMuT may vary from system to system. However, this also means that the deterministic IRs in the RQ4 experiment performed as well as both the deterministic and probabilistic IRs in the other experiments, which suggests that IR types that remain present may still be able to produce comparably accurate classifications.

Another interesting observation is that substantially fewer IRs are used in the RQ4 experiment in comparison to the RQ1 and RQ2 experiments; we only constructed four IRs for Dijkstra's Algorithm compared to 48 for the Genetic Algorithm. This shows that IMuT can be effective with a very small number of IRs which supports the findings of our subsumption analysis (see Section 5.3.1.1). It also illustrates that the number of IRs in the test set may not positively correlate with accuracy.

5.3.5 RQ5: Can Our Findings Revolving Around the Productivity Gains Offered by IMuT Generalise to Other Problem Domains?

In this section, we investigate whether our findings regarding the productivity gains offered by IMuT can generalise to other systems — RQ5. To do this, we obtained three subject programs (Bubble Sort, Binary Search, and Knuth-Morris-Pratt) and three test suites that each consist of 100 test cases (one for each subject program), generated 30 non-equivalent mutants across these programs and retained all equivalent mutants were discovered while acquiring these 30 non-equivalent mutants; 4 equivalent mutants were found across the three subject programs. Further details about the experimental design can be found in Section 5.2.

It's not surprising that all four equivalent mutants were correctly classified because all of the IRs were Deterministic IRs. These results are consistent with our observations for the Genetic Algorithm subject program and so suggest that our observations regarding equivalent mutants will generalise to broader types of systems. For example, the user doesn't have to manually check any of the non-equivalent mutant classifications for Deterministic IRs, because none of the equivalent mutants are likely to have been misclassified as non-equivalent by these IRs. Only Deterministic IRs were leveraged for these subjects because we were unable to identify Probabilistic IRs for these subject programs. This supports our previous observation that certain IR types may not be available in some systems.

All 30 non-equivalent mutants were also correctly classified. Since all non-equivalent mutant classifications suggested by Deterministic IRs can be trusted, the user only needs to manually inspect the 4 equivalent mutants. Thus, the total number of mutants that must be manually inspected by the user is reduced by 88.24%. These productivity gains (as measured by the number mutants that must be manaully inspected and the number of mutants that do not have to be manually inspected) are significantly higher than those reported for RQ2 (Fisher's Exact Test: $p < 0.05$). This is because the IRs used for RQ5 performed better, and because MuJava generated proportionally fewer equivalent

mutants; if more equivalent mutants had been generated, then the user would have had to inspect more mutants, since equivalent mutant classifications cannot be trusted. This is consistent with the findings of Zeller and Schuler [203] who also found that mutant samples that are composed of a greater proportion of equivalent mutants require more manual inspection effort. This demonstrates that the productivity gains offered by IMuT can vary in different subject programs. Regardless, the improvements were substantial in all cases, which suggests that the technique can add value for different types of systems.

5.4 Related Work

5.4.1 The Equivalent Mutant Problem and Non-Deterministic Systems

Non-deterministic systems are becoming increasingly prevalent e.g. concurrency can lead to alternative interleavings. For example, consider a variable X that is instantiated with a value of 3. Suppose we have two threads t_1 and t_2 and that t_1 applies the following operation to X: $X = X + 1$. Further, suppose that t_2 updates the value of X to $X = X \times 2$. The order of the interleavings affects the final state of X i.e. if t_1 executes first, then $X = 8$ and if t_2 executes first $X = 7$.

This complicates the mutant classification process. Several proposals have been made to address this. For example, Carver [21] identifies two methods - Multiple Execution Testing (MET) and Deterministic Execution Testing (DET). In MET, confidence is improved by executing the original S_o and mutant S_m versions multiple times and observing their output distributions. DET involves forcing the SUT to execute deterministically by manipulating conditions e.g. a Genetic Algorithm's Mutation Rate can be set to 100% or 0% to force deterministic execution of the Mutation Operator.

Both strategies are viable, but have limitations. For example, MET is dictated by chance; thus there is scope for misclassification. It's also expensive because it uses multiple executions. On the other hand, DET limits test case selection; thus some mutation points may not be reachable with allowable test cases. Carver [21] attempted to reduce the impact of these weaknesses by combining MET and DET.

Gligoric et al. [65] suggest executing S_o with a test case t, and then establishing whether the mutant statement in S_m could have been reached by this execution. Non-reachability implies equivalence for t. This approach is limited to the identification of equivalent mutants in unexecuted code.

Finally, Papadakis et al. [152] propose comparing the object code of S_m to the object code of S_o. If the object code of S_m matches the object code of S_o, then we can guarantee that S_o is equivalent to S_m. However, if the comparison reveals that there are discrepancies, S_m may either be equivalent or non-equivalent to S_o. Although the approach can't correctly classify all mutants, it is inexpensive and so can be a valuable complementary equivalent mutant classification technique.

5.4.2 Weak Mutation Testing

Let S be the system. Suppose that mutation testing was applied to S to obtain a mutant M. Let m denote the point in M that the mutation was inserted, and s be the corresponding point in S. Weak Mutation Testing involves executing S and M with the same test case and comparing the state that immediately manifests after s with the state that immediately manifests with m [86]. Discrepancies

are interpreted to be an indication that M is not equivalent to S, and M is otherwise considered to be equivalent to S if the converse is true.

Weak Mutation Testing is similar to IMuT, in that both techniques classify mutants based on internal state information, but there are some noteworthy differences. For example, since Weak Mutation Testing immediately checks the program state that manifests after m, it may be more likely to correctly classify non-equivalent mutants than IMuT. However, this also means that it could be more likely to misclassify equivalent mutants.

Consider another example; an oracle in Weak Mutation Testing is associated with one point in the program, and can only provide classification support for mutants that have mutated this point in the program. Thus, one must develop one oracle per distinct mutation point. By contrast, an IR in IMuT is not restricted to one program point, and so can potentially provide classification support for a wider range of mutants. The implication of this disparity are as follows. IMuT has a larger initial cost because of the IR development task that must be carried out before mutation testing is undertaken, but this cost is fixed. By contrast, there is a marginal cost associated with each distinct mutation point in Weak Mutation Testing because an oracle must be developed for each distinct mutation point. Thus, if a large number of mutants are required in an experiment and there are a large enough number of distinct mutation points among these mutants, the cost of Weak Mutation Testing can exceed IMuT's cost. To that end, the most cost effective choice of technique is context dependent.

Another noteworthy difference includes the following. IMuT only requires one to execute M, whilst Weak Mutation Testing necessitates the execution of S and M. Thus, IMuT might be cheaper to apply because it requires fewer executions. However, IMuT requires one to execute the entire program, whilst Weak Mutation Testing can terminate an execution after the mutation point has been reached [86]. Thus, an execution in IMuT might be more expensive; this may offset the cost savings that are accrued from requiring fewer executions.

5.5 Threats to Validity

All of the threats to validity that were outlined in Section 3.5 are relevant to the experiments conducted in this chapter.

Recall that TEMDT was applied to the Genetic Algorithm subject program based on the output solution, and its fitness value. Mutant classifications based on the fitness value, and thus the validity of our results concerning the effectiveness of TEMDT when it is only applied based on the fitness value, may have been affected by the real fault. However, this would not have changed our conclusions regarding the overall effectiveness of TEMDT, since mutant classifications based on the output solution were not affected by the real fault and led to the misclassification of all of the mutants.

We used a refactoring tool to generate some of the equivalent mutants, instead of using MuJava. Since the changes made by the refactoring tool may be different to the changes made by MuJava, it could be possible that our technique may have performed differently, had MuJava been used. However, we observed that IMuT obtained a comparable level of effectiveness for equivalent mutants produced by both of the tools, which suggests that the use of the refactoring tool did not have a meaningful impact on the results.

In Section 4.4, we noted that the results concerning the effectiveness of our IRs in Chapters 3 and 4 were comparable. Encouragingly, we observe that the effectiveness of our IRs in this chapter are also comparable to these results. This reinforces our supposition that the mutant sample was large enough to characterise the typical distribution of faults that can and can't be detected by our IRs.

5.6 Conclusion

In this chapter, we proposed Interlocutory Mutation Testing, a mutant classification technique that can be applied in the presence of coincidental correctness and/or non-determinism. The technique correctly classified 93.33% of the non-equivalent mutants and 100% of the equivalent mutants despite the presence of coincidental correctness and non-determinism, which suggests that the technique is capable of producing highly accurate results in this context. This indicates that Objective 3 has been addressed (see Section 1.1). We also determined that IMuT can reduce the manual effort required to determine the equivalence of mutants by 63.33%, and can thus improve productivity. Finally, we explored the generalisability of these results and discovered that these findings are likely to generalise to other systems.

We also compared our technique to a widely used technique that we refer to as Traditional Equivalent Mutant Detection Technique (TEMDT). In TEMDT, the original version of the SUT, S_o, and mutant version, S_m, are executed with a test suite to obtain a set of pairs $\langle S_o(O), S_m(O)\rangle$, where $S_o(O)$ and $S_m(O)$ are the outputs of S_o and S_m respectively. TEMDT assumes the following: if each $S_o(O) = S_m(O)$, then S_o and S_m are equivalent. We found that IMuT was able to substantially outperform TEMDT. During this comparison we uncovered compelling evidence that indicates that mutant classification accuracy might be improved by restricting the mutation testing tool to certain subsets of the output. We believe that this finding may generalise to other mutation testing tools and techniques and thus suggest investigating this possibility in future research.

One limitation of our work is the effort required to apply the technique. Our experiments that were based on the Genetic Algorithm subject program leveraged 48 IRs, which may be unacceptable in some cases. In Section 5.3.1.1, we observed that a small proportion (10) of these IRs subsumed all of the other IRs. We also observed that only 4, 1, 1, and 3 IRs were required to obtain a similar level of effectiveness in the other subject programs. This demonstrates that the technique can be applied with relatively few IRs (which may be more acceptable in the aforementioned cases), if one restricts their development effort to a small set of effective IRs. Unfortunately, the results did not indicate how one might do this. We would therefore like to investigate this in future work.

In Section 3.3.6.2, we detailed the partially automated process that is used to develop IRs. Increasing the degrees of automation further will also reduce the effort required to use the technique and so can reduce the impact of the limitation above. Thus, for future work, we would like to explore methods of automating the development of IRs further.

In summary, this chapter introduced a partial solution to the Equivalent Mutant Problem, that can tolerate coincidental correctness and non-determinism, and therefore satisfied Objective 3. The nature of this partial solution was a problem domain specific methodology for applying Interlocutory Testing. In the next chapter, we investigate how Interlocutory Testing can be integrated into Spectrum-based Fault Localisation to mitigate the impact of coincidental correctness on Spectrum-based Fault Local-

isation.

Chapter 6

Interlocutory Spectrum-based Fault Localisation

Spectrum-based Fault Localisation (SBFL) is a debugging technique that uses the coverage information in passed and failed test cases to determine the likelihood that a program statement is faulty. As discussed in Section 1.1.4, coincidental correctness can reduce the effectiveness of traditional testing techniques; thus failed test cases can be mislabelled as passed. This mislabelling can compromise the effectiveness of SBFL. Section 2.8 revealed that coincidental correctness is widespread, and thus indicates that this issue is ubiquitous. This motivated us to develop a solution for this problem. Chapter 3 introduced Interlocutory Relations, which are test oracles based on Interlocutory Testing, and demonstrated their effectiveness for testing in the presence of coincidental correctness. In this chapter, we introduce Interlocutory Spectrum-based Fault Localisation (ISBFL); an extended version of SBFL, that incorporates Interlocutory Relations to tolerate coincidental correctness. Thus, this chapter attempts to address Objective 4 (see Section 1.1).

The main contributions of this chapter are:

1. A new SBFL technique called ISBFL that extends the generalisability of SBFL to systems with coincidental correctness.
2. An evaluation of the fault localisation effectiveness of ISBFL.
3. A comparative analysis between ISBFL and three well-known SBFL techniques — Tarantula, Ochiai and Jaccard.

Sections 2.8 and 3.4 presents relevant background material for this chapter, and Section 6.1 introduces ISBFL. In Section 6.2, we describe the experiments that were conducted to evaluate ISBFL, and discuss the results of these experiments in Section 6.3. Threats to validity are outlined in Section 6.4, and conclusions are finally drawn in Section 6.5.

6.1 Interlocutory Spectrum-based Fault Localisation — Technique Description

As mentioned above, SBFL (which was introduced in Section 1.1.4) can be negatively affected by coincidental correctness, and Interlocutory Testing (which was introduced in Section 3.1) is an effective means of suppressing the effects of coincidental correctness. Therefore, one solution for mitigating the impact of coincidental correctness on SBFL might be the integration of SBFL with Interlocutory Testing. We call such a solution ISBFL.

Recall that SBFL techniques leverage the coverage information in passed and failed test cases to determine the likelihood that a program statement is faulty. Also recall that Interlocutory Testing has two types of IRs — Deterministic and Probabilistic IRs. Probabilistic IRs judge the correctness of the entire SUT, based on execution trace data from across all of the test cases in the test suite. This means that such IRs cannot distinguish between passed and failed test cases. Since Deterministic IRs judge the correctness of the SUT based on execution trace data that is available in a single test case, they do not suffer from this limitation. These differences necessitate different strategies for the integration of SBFL with Interlocutory Testing, depending on the types of IRs that are used. In this section, we introduce two variants of ISBFL; one is based on Deterministic IRs (we call this variant of ISBFL Deterministic ISBFL (DISBFL)), and the other is based on Probabilistic IRs (this variant of ISBFL is called Probabilistic ISBFL (PISBFL)). The former is described in Section 6.1.1, and the latter is presented in Section 6.1.2.

6.1.1 Interlocutory Spectrum-based Fault Localisation: Deterministic IRs

This section draws on the example used in Section 3.1. To reiterate, the system in this example is a Genetic Algorithm that is composed of four major components: Initial Population Generator, Crossover, Mutation, and Selection. $Population_{SOI}$ and $Population_{SOO}$ denote the input and output of the Selection component respectively, and $CrossoverN$ quantifies the number of solutions that were generated by the Crossover component. IR_1 is an IR that predicts that $CrossoverN == Population_{SOI}.size() - Population_{SOO}.size()$, when $Population_{SOI}.size() > Population_{SOO}.size()$.

6.1.1.1 Intuition

The process of applying SBFL can be broken down into four major steps:

1. Let $ts = \{tc_1, tc_2, ..., tc_n\}$ be a test suite that consists of n test cases. The following procedure can be used to process one test case $tc_i \in ts$. tc_i is executed, and the distinct set of program statements that executed during the execution of this test case is recorded. Let $Spectra_i$ denote this set of program statements. Each $stmt_j \in Spectra_i$ was executed before $stmt_{j+1} \in Spectra_i$. SBFL applies this procedure to each test case in ts, to produce a set $Spectras = \{Spectra_1, Spectra_2, ..., Spectra_n\}$, where $Spectra_i \in Spectras$ corresponds to $tc_i \in ts$. This constitutes the first step of the SBFL process.

2. The second step of the process involves leveraging an arbitrary testing technique to classify each $tc_i \in ts$ as passed or failed.

3. During the third step of the process, SBFL computes $PSpectras$ to be a subset of $Spectras$, such that each $Spectra_i \in PSpectras$ is associated with a passed test case. $FSpectras$ is computed to be a subset of $Spectras$, such that each $Spectra_i \in FSpectras$ is associated with a failed test case.

4. The final step consists of SBFL computing a suspiciousness score for each program statement $stmt_j$ in the program, that is based on the frequency with which $stmt_j$ appears in $PSpectras$ and $FSpectras$.

Interlocutory Testing can be integrated into SBFL by altering the process described above as follows. In the first step of the process, during the execution of a given test case tc_i, the step can be modified to additionally capture the execution trace LOG_i of tc_i. Thus, in addition to $Spectras$, the first step of the process also produces a set $LOGs = \{LOG_1, LOG_2, ..., LOG_n\}$, such that $LOG_i \in LOGs$ corresponds to $tc_i \in ts$. The second step of the process can be replaced with the following iterative procedure. For each test case $tc_i \in ts$, let $Spectra_i$ and LOG_i be the distinct set of program statements that executed during tc_i, and the execution trace of tc_i respectively. IRs are evaluated based on LOG_i; if at least one IR fails, then tc_i is classified as failed, or is otherwise classified as passed. The results of these IR evaluations provide some evidence regarding the location of the fault. This evidence is finally used to refine $Spectra_i$. The intuition behind DISBFL is to combine Interlocutory Testing with SBFL in this manner.

6.1.1.2 Technique Description

Algorithm 8: Deterministic Interlocutory Spectrum-based Fault Localisation

Input: $TestSuite$

Output: $SuspiciousnessScores$

1 Let $ProgramSpectras_p$ and $ProgramSpectras_f$ be empty lists;
2 **foreach** $tc_i \in TestSuite$ **do**
3   The SUT is executed with tc_i. Let $ExecutionTrace_i = \langle State_1, State_2, ..., State_n \rangle$ denote the execution trace;
4   A set of IRs IRs are evaluated (as described in Section 3.1.1) based on $ExecutionTrace_i$;
5   An IRInstance, $IRInstance_x$, is a pair $\langle IR_y, States_x \rangle$, such that $States_x$ is a subset of $ExecutionTrace_i$ and execution trace data was extracted from all states in $States_x$ for an evaluation of $IR_y \in IRs$. Let $IRInstances_i$ be the set of all IRInstances in $ExecutionTrace_i$;
6   Let $FailedIRInstances_i$ be an empty list;
7   **foreach** $IRInstance_a = \langle IR_y, States_a \rangle \in IRInstances_i$ **do**
8   Let $LastState_a \in States_a$, such that all other program states in $States_a$ manifested earlier than $LastState_a$. $IRInstance_a$ is associated with a distinct set of program statements, such that these program statements executed before $LastState_a$ manifested;
9         $IRInstance_a$ is also associated with the pass/fail verdict that was determined by the evaluation of $IR_y.evaluateIR(States_a)$;
10   **if** $IRInstance_a$ *is associated with a fail verdict* **then**
11            $FailedIRInstances_i.add(IRInstance_a)$;
12   **end**
13   **end**
14   **if** $FailedIRInstances_i.isEmpty()$ **then**
15   Let $ProgramSpectra_i$ be a distinct set of program statements, such that these program statements executed at least once during the execution of tc_i;
16        $ProgramSpectras_p.add(ProgramSpectra_i)$;
17   **else**
18   Let $ProgramSpectra_i$ be the set of program statements that are associated with $FailedIRInstance_i$, such that $FailedIRInstance_i$ is the IRInstance in $FailedIRInstances_i$ that is associated with the fewest program statements;
19        $ProgramSpectras_f.add(ProgramSpectra_i)$;
20   **end**
21 **end**
22 Let $SuspiciousnessScores$ be an empty set;
23 **foreach** *program statement s in the SUT* **do**
24     $SuspiciousnessScores.add(\langle s, computeSuspiciousnessScore(s, ProgramSpectras_p, ProgramSpectras_f) \rangle)$;
25 **end**
26 $SuspiciousnessScores$ is sorted in descending order, based on the suspiciousness scores;

Algorithm 8 describes how DISBFL realises the intuition above. This section explains Algorithm 8 in detail.

Let Sys be the SUT, and tc_i be a test case for Sys. Lines 3 – 20 of Algorithm 8 outline DISBFL's procedure for obtaining the program spectra of tc_i. The first step of this procedure involves executing Sys with tc_i (Line 3 of Algorithm 8). The Execution Trace of tc_i is a sequence of program states $ExecutionTrace_i = \langle State_1, State_2, ..., State_n \rangle$, such that each $State_j \in ExecutionTrace_i$ manifested during the execution of tc_i, and each $State_j$ manifested before $State_{j+1}$. Let IRs be a set of IRs that were developed for Sys. The second step of the procedure consists of evaluating IRs based on $ExecutionTrace_i$ (as described in Section 3.1.1).

An IRInstance, $IRInstance_x$, is a pair $\langle IR_y, States_x \rangle$; $States_x$ is a subset of the set of states that appear in $ExecutionTrace_i$, such that execution trace data was extracted from all states in $States_x$ for an evaluation of $IR_y \in IRs$. To illustrate, consider the Genetic Algorithm program example. Suppose that this program was executed with test case tc_{ex}, which performed two generations, and produced the following Execution Trace: $ExecutionTrace_{ex} = \{State_1, State_2, ..., State_n\}$. Suppose that states $State_{51}$ to $State_{100}$ correspond to program states that manifested during the first generation, and that $State_{60}$, $State_{70}$ and $State_{75}$ correspond to program states that contain execution trace data pertaining to $CrossoverN$, $Population_{SOI}$ and $Population_{SOO}$ respectively. Suppose that this execution trace data was extracted from these three program states, for an evaluation of IR_1, as described in Section 3.1.1.2. Thus, an IRInstance would be $IRInstance_{ex1} = \langle IR_1, States_{ex1} \rangle$, such that $States_{ex1}$ consists of $State_{60}$, $State_{70}$ and $State_{75}$.

Since IR_y may be evaluated multiple times with execution trace data from different states, or multiple IRs in IRs may be evaluated in tc_i, there may be multiple IRInstances in $ExecutionTrace_i$. To illustrate the former case, consider the previous example; suppose that states $State_{101}$ to $State_{150}$ correspond to program states that manifested during the second generation, and that $State_{110}$, $State_{120}$ and $State_{125}$ correspond to program states that contain execution trace data pertaining to $CrossoverN$, $Population_{SOI}$ and $Population_{SOO}$ respectively. Let us suppose that this execution trace data was extracted from these three program states, for another evaluation of IR_1. Thus, another IRInstance would be $IRInstance_{ex2} = \langle IR_1, States_{ex2} \rangle$, such that $States_{ex2}$ consists of $State_{110}$, $State_{120}$ and $State_{125}$. Line 5 of Algorithm 8 defines $IRInstances_i$ to be the set of all IRInstances in $ExecutionTrace_i$.

```
1.  y = 0
2.  for (i = b; i <5; i++)
3.      y = y + 1
4.  z = x + y
5.  a = z - 2
```

Figure 6.1: Sample Program Fragment

Lines 7 – 13 iterate over each IRInstance in $IRInstances_i$. On a given iteration, $IRInstance_a = \langle IR_y, States_a \rangle$ denotes the IRInstance being considered on this iteration. Three noteworthy operations are performed during an iteration. Let $LastState_a$ be a program state in $States_a$, such that $LastState_a$ manifested at a later point in the execution trace than all other states in $States_a$. For example, the $LastState_{ex1}$ of $IRInstance_{ex1}$ would be $State_{75}$, because $State_{75}$ manifests after $State_{60}$ and $State_{70}$. Line 8 of Algorithm 8 performs the first noteworthy operation; it associates $IRInstance_a$ with the

distinct set of program statements that executed before $LastState_a$ manifested. To illustrate, suppose that the sample program in Figure 6.1 was executed with a test case tc_{sample} that set $b = 0$, and two IRs were evaluated during the execution — let $IRInstance_{sample1}$ and $IRInstance_{sample2}$ be the IRInstances that are based on these evaluations. Since the condition for leaving the for loop was not immediately satisfied, all program statements in the sample program would have been executed. Let us suppose that the state that manifested after the execution of statement 4 is the $LastState_{sample1}$ of $IRInstance_{sample1}$, and that the state that manifested after the execution of statement 5 is the $LastState_{sample2}$ of $IRInstance_{sample2}$. This means that $IRInstance_{sample1}$ and $IRInstance_{sample2}$ are both associated with statements 1 to 4, and $IRInstance_{sample2}$ is additionally associated with statement 5.

During the execution of Line 4 of Algorithm 8, $IRInstance_a$'s IR_y was evaluated with execution trace data from $IRInstance_a$'s $States_a$ and this would have produced a pass/fail verdict. Line 9 of Algorithm 8 implements the second noteworthy operation; the line associates $IRInstance_a$ with this verdict. The final noteworthy operation is performed by Lines 6 and 10 – 12. These lines are used to define $FailedIRInstances_i$ to be a subset of $IRInstances_i$, such that $FailedIRInstances_i$ only contains IRInstances that are associated with a fail verdict.

tc_i is deemed to have failed if $FailedIRInstances_i \neq \emptyset$, or is otherwise considered to have passed. Lines 14 – 20 then adopt one of two methodologies for computing the program spectra of tc_i; the exact choice of methodology depends on whether tc_i passed or failed. In particular, if tc_i is a failed test case, then its associated $ProgramSpectra_i$ is the set of statements that are associated with $FailedIRInstance_i$, such that $FailedIRInstance_i$ is the IRInstance in $FailedIRInstances_i$ that is associated with the fewest program statements. In continuation of the previous example, suppose that $IR_{sample1}$ and $IR_{sample2}$ both failed, and therefore tc_{sample} failed. Since $IRInstance_{sample1}$ is associated with fewer lines of code than all other IRInstances that are associated with a failed verdict i.e. $IRInstance_{sample2}$, the program spectra $ProgramSpectra_{sample}$ of tc_{sample} will consist of the program statements that are associated with $IRInstance_{sample1}$ — in the case of the running example, statements 1 to 4.

However, if tc_i is a passed test case, then its associated $ProgramSpectra_i$ is the set of all distinct program statements that executed at least once, during the execution of tc_i. In continuation of the example above, if tc_{sample} passed, then its program spectra, $ProgramSpectra_{sample}$, would consist of the following statements: statements 1 to 5.

To briefly recap, the entire discussion above pertains to Lines 3 – 20 of Algorithm 8, and outlines the procedure that is used by DISBFL for computing the program spectra of a test case. The input for Algorithm 8 is a test suite $TestSuite$, which is a set of test cases. The procedure defined across Lines 3 – 20 is applied to each test case in $TestSuite$; Line 2 of Algorithm 8 facilitates this. Thus, one program spectra will have been computed for each test case in $TestSuite$. Let $ProgramSpectras_p$ be a subset of these program spectras, such that each program spectra in this subset was computed based on a passed test case. Similarly, let $ProgramSpectras_f$ be a subset of these program spectras, such that each program spectra in $ProgramSpectras_f$ was computed based on a failed test case. Lines 1, 16, and 19 of Algorithm 8 are used to compute these subsets.

The suspiciousness score of a program statement is a numerical value that quantifies the likelihood of that program statement being faulty. Let $Failed_s$ be the number of program spectras in

$ProgramSpectras_f$ that program statement s appears in. Similarly, let $Passed_s$ be the number of program spectras in $ProgramSpectras_p$ that s appears in. The Ochiai Formula is a method of computing the suspiciousness score of s. The Ochiai Formula [52] is:

$$Ochiai = Failed_s \div \sqrt{(Passed_s + Failed_s) \times ProgramSpectras_f.size()}$$

DISBFL uses the Ochiai formula in conjunction with $ProgramSpectras_p$ and $ProgramSpectras_f$, to compute the suspiciousness score of each program statement in Sys. This is achieved by Lines 22 – 25.

To illustrate, suppose that the SUT is the program in Figure 6.1, and that the SUT is faulty. In particular, statement 3 should be "y = y + 2". Also suppose that the SUT is associated with one IR, and that this IR results in exactly one IRInstance $IRInstance_{example}$ regardless of which test case is used. The $LastState_{example}$ of $IRInstance_{example}$ is the program state that always manifests after the execution of statement 4. Finally, let us suppose that $TestSuite$ consists of three test cases tc_{p1}, tc_{p2} and tc_f. tc_{p1} and tc_{p2} sets $b = 6$ and $b = 7$ respectively, and are passed test cases, thus, the $ProgramSpectra_{p1}$ of tc_{p1} and the $ProgramSpectra_{p2}$ of tc_{p2} consist of the following statements: 1, 2, 4 and 5. tc_f sets $b = 3$ and causes the IR to fail; thus tc_f is a failed test case, and its $ProgramSpectra_f$ is associated with statements 1, 2, 3 and 4. To compute the suspiciousness score of statement 1, we first obtain $Failed_1 = 1$, since it appears in only one failed test case's program spectra. We also ascertain $ProgramSpectras_f.size() = 1$, because there is only one failed test case. Similarly, we obtain $Passed_1 = 2$ because statement 1 executes in two passed test cases, and $ProgramSpectras_p.size() = 2$ because there are two passed test cases in total. Using the Ochiai formula, we obtain the following suspiciousness score for statement 1: 0.58. Repeating this process for the remaining statements 2, 3, 4 and 5 yields the following respective suspiciousness scores: 0.58, 1, 0.58 and 0. The faulty statement has been awarded the highest suspiciousness score.

Finally, all program statements in Sys are sorted in descending order of suspiciousness by Line 26.

6.1.2 Interlocutory Spectrum-based Fault Localisation: Probabilistic IRs

6.1.2.1 Intuition

Recall that Probabilistic IRs judge the correctness of the system based on the entire test suite, which means that these IRs cannot distinguish between passed and failed test cases. However, despite this, it is possible to distinguish between passed and failed Probabilistic IRs. We therefore reasoned that it might be possible to create IR specific program spectra, instead of test case specific program spectra, and use this new type of spectra for fault localisation. This is the intuition behind PISBFL.

6.1.2.2 Technique Description

Algorithm 9: Probabilistic Interlocutory Spectrum-based Fault Localisation

Input: $TestSuite$

Output: $SuspiciousnessScores$

1 The SUT is executed with $TestSuite$ to obtain a set of execution traces $ExecutionTraces$;
2 A set of Probabilistic IRs IRs are evaluated based on $ExecutionTraces$, as described in Section 3.1.2. Let IRs_f be the set of all Probabilistic IRs that reported a failure;
3 Let $ProgramSpectras$ be an empty list;
4 **foreach** $IR_i \in IRs_f$ **do**
5   Let $PartialSpectras_i$ be an empty list;
6   **foreach** $ExecutionTrace_j \in ExecutionTraces$ **do**
7     $PartialSpectra^i_j = getPartialSpectra(IR_i, ExecutionTrace_j)$;
8     $PartialSpectras_i.add(PartialSpectra^i_j)$;
9   **end**
10   Let $ProgramSpectra_i$ be the set of distinct program statements across all $PartialSpectra^i_j \in PartialSpectras_i$;
11   $ProgramSpectras.add(ProgramSpectra_i)$;
12 **end**
13 Let $SuspiciousnessScores$ be an empty set;
14 **foreach** *program statement s in the SUT* **do**
15   Let $SuspiciousnessScore$ denote the number of program spectras in $ProgramSpectras$ that s appears in;
16   $SuspiciousnessScores.add(\langle s, SuspiciousnessScore \rangle)$;
17 **end**
18 $SuspiciousnessScores$ is sorted in descending order, based on the suspiciousness scores;

Algorithm 9 describes the procedure for conducting ISBFL with probabilistic IRs, and therefore explains how the intuition above is implemented.

Let Sys be the SUT, IRs be a set of Probabilistic IRs that were developed for Sys and $TestSuite$ be a set of test cases. Algorithm 9 first executes Sys with $TestSuite$, to obtain a set of execution traces $ExecutionTraces$ (Line 1 of Algorithm 9). In PISBFL, each $IR_i \in IRs$ is evaluated based on $ExecutionTraces$, as described in Section 3.1.2. This is achieved by Line 2 of Algorithm 9. Let IRs_f be the set of Probabilistic IRs that reported a failure.

Let $getPartialSpectra(IR_i, ExecutionTrace_j)$ be a function that accepts a Probabilistic IR, IR_i, and an Execution Trace, $ExecutionTrace_j$, as input. The function operates as follows. It first obtains all of the IRInstances, $IRInstances_j$, from $ExecutionTrace_j$. It then computes $IRInstances^i_j$ to be a subset of $IRInstances_j$, such that for each $IRInstance_a = \langle IR_y, States_a \rangle$ in $IRInstances^i_j$, $IR_y = IR_i$. Let $IRInstance_i$ be the IRInstance in $IRInstances^i_j$ that is associated with the most program statements. $PartialSpectra^i_j$ is the set of statements that are associated with $IRInstance_i$. $PartialSpectra^i_j$ is the output of the function.

Lines 5 – 10 of Algorithm 9 outline a two step process that can be used to compute the program spectra $ProgramSpectra_i$, of a specific IR_i (e.g. TournamentPIR from Section 3.1.2). Lines 6 – 9 of Algorithm 9 describe the first step of this process: for each $ExecutionTrace_j \in ExecutionTraces$, the

function described above is invoked, such that the input to the function is IR_i and $ExecutionTrace_j$, to obtain $PartialSpectra_j^i$. Thus, on each iteration, we obtain a $PartialSpectra_j^i$, that corresponds to the $ExecutionTrace_j \in ExecutionTraces$ being considered on that iteration, and IR_i which is considered on all iterations. Lines 5 and 8 of Algorithm 9 are used to define $PartialSpectras_i$ to be a set of all $PartialSpectra_j^i$ that were produced across these iterations. The second step of the process is implemented by Line 10 of Algorithm 9; the step defines the program spectra, $ProgramSpectra_i$, of IR_i to be the distinct set of program statements across all $PartialSpectra_j^i \in PartialSpectras_i$.

By means of Lines 3, 4, and 11, PISBFL leverages the two step process described above to compute a program spectra, $ProgramSpectra_i$, for each $IR_i \in IRs_f$, and defines $ProgramSpectras$ to be the set of these program spectras. The suspiciousness score of a program statement s is a count of the number of program spectras in $ProgramSpectras$ that s appears in. Lines 13 − 17 of Algorithm 9 determine the suspiciousness score of each program statement in the SUT, and Line 18 finally sorts all of the program statements in Sys in descending order of suspiciousness.

6.1.3 Applying Interlocutory Spectrum-based Fault Localisation

Situations may exist, in which one has access to both Deterministic and Probabilistic IRs. In such situations, ISBFL first applies DISBFL to compute the suspiciousness of each line of code. If at least one program statement is associated with a suspiciousness score that is greater than 0, ISBFL returns these suspiciousness scores as its output. Otherwise, it then applies PISBFL to compute the suspiciousness scores, and returns these suspiciousness scores as the output.

In other words, ISBFL only leverages PISBFL, when DISBFL fails to provide any fault localisation support. The rationale for prioritising DISBFL over PISBFL is two-fold. Firstly, unlike PISBFL, DISBFL is not susceptible to false positives. Secondly, we suspect that DISBFL is likely to provide better fault localisation, because it can capitalise on the distinction between passed and failed test cases, unlike the PISBFL.

6.2 Experimental Design

6.2.1 Research Questions

In this chapter, we attempt to address the following research questions:

RQ1 Is ISBFL a feasible[1] debugging technique? This research question assesses the feasibility of ISBFL.

RQ2 How effective is ISBFL at localising faults? To address this research question, we explore the extent to which ISBFL can minimise the number of LOC that must be manually inspected to find a fault.

RQ3 What are the factors that affect the fault localisation effectiveness of ISBFL? We investigate factors that may correlate with the effectiveness of the ISBFL.

[1]In the context of this research question, feasibility refers to whether the technique is capable of carrying out its designated task.

RQ4 How does the fault localisation effectiveness of ISBFL compare to other SBFL techniques? We addressed this research question by comparing the fault localisation effectiveness of ISBFL with three well-known SBFL techniques — Tarantula, Ochiai, and Jaccard.

Two different experiments were conducted to answer these research questions. The first addresses RQ1, and second addresses RQ2 — RQ4. The remainder of this section presents the experimental design for these experiments.

6.2.2 Subject Programs

We leveraged four subject programs across the two experiments. In particular, we used the Dijkstra's Algorithm, Bubble Sort, and Knuth-Morris-Pratt subject programs that were described in Section 3.2.3.1 in the experiment that was designed to address RQ1. The Genetic Algorithm subject program that was described in Section 3.2.1.1 was used in the experiment for RQ2 — RQ4. Our justifications for the use of these subject programs can be found in Section 3.2.3.1 and Section 3.2.1.1.

6.2.3 Test Cases

100 test cases were obtained for each subject program described in Section 6.2.2. Thus, a total of 400 test cases were used throughout our experiments. The test cases for the Dijkstra's Algorithm, Bubble Sort, and Knuth-Morris-Pratt subject programs were generated using the same test case generation strategies that were detailed in Section 3.2.3.3. We used these test case generation strategies for the reasons outlined in Section 3.2.3.3. In this experiment, the Genetic Algorithm subject program used the same 100 test cases that were used in Chapter 3. This test suite was used for the reasons described in Section 3.2.1.3.

6.2.4 Faults

We used the same mutant generation strategies that were outlined in Sections 3.2.3.2 and 3.2.1.2 for the RQ1 subject programs, and RQ2 — RQ4 subject program respectively (see Sections 3.2.3.2 and 3.2.1.2 for justifications regarding these decisions). A total of 90 mutants were generated. 30 mutants were generated across the RQ1 subject programs (10 each), and 60 mutants were generated for the RQ2 — RQ4 subject program. 38 of these mutants were coincidentally correct, and 22 were standard.

6.2.5 Interlocutory Relations

We leveraged the same 56 IRs that were used in Chapter 3. In particular, 48, 4, 1, and 3 IRs were used for the Genetic Algorithm, Dijkstra's Algorithm, Bubble Sort, and Knuth-Morris-Pratt subject programs respectively. A list of these IRs can be found in Appendices A, C, D and F respectively. Each IR is also associated with a summary of the main aspects of that IR. We envisage a scenario in which one applies ISBFL after one has completed testing by means of either Interlocutory Testing or Interlocutory Metamorphic Testing. Realistically, in such a scenario, all of the IRs that were utilised during testing would be leveraged in ISBFL. Thus, our decision to include all of the IRs that were leveraged by Interlocutory Testing/Interlocutory Metamorphic Testing in Chapter 3/Chapter 4 was partly motivated by the opportunity to improve the representativeness of the experiment, with

respect to this scenario. Our decision was also motivated by the fact that this collection of IRs was large enough to support the types of analysis that were conducted in this chapter.

6.2.6 Measures

The EXAM Score is a measure of the number of lines of code that must be manually inspected, before the faulty line has been found [191]. We use three variants of this measure for our experiments. Let $MoreSuspicious$ and $EquallySuspicious$ be two sets of statements, such that all of the lines of code in $MoreSuspicious$ and $EquallySuspicious$ are more and equally suspicious than the faulty statement respectively. $EquallySuspicious$ also contains the faulty line of code. Since the user must check each line of code in order of suspiciousness, they will first manually inspect all of the lines of code in $MoreSuspicious$ (and will not find the fault). Afterwards, they must inspect the lines of code in $EquallySuspicious$. In the best case, the first line of code that they inspect in $EquallySuspicious$ may be the faulty line. Thus, we define a metric, EXAMBest, as follows: $EXAMBest = MoreSuspicious.size() + 1$. In the worst case, the last line of code they inspect in $EquallySuspicious$ may be the faulty line. We define another metric, EXAMWorst to be: $EXAMWorst = MoreSuspicious.size() + EquallySuspicious.size()$. Finally, in the average case, the tester will have to inspect approximately half of $EquallySuspicious$, thus we define a final metric, EXAMAverage to be: $EXAMAverage = MoreSuspicious.size() + (EquallySuspicious.size() \div 2)$. See Section 6.4.3 for our motivations for adopting these measures.

6.2.7 Benchmark Techniques

We leverage three benchmark techniques. All three of these benchmarks use the standard SBFL process outlined in Section 1.1.4, but each one uses a different suspiciousness score formula. The benchmarks are: Tarantula, Ochiai, and Jaccard. The Ochiai Formula was introduced in Section 6.1.1. The formula for Tarantula and Jaccard are defined as follows [52, 89]. Let $TotalPassed$ and $TotalFailed$ be the total number of passed and failed test cases respectively, and $Passed_s$ and $Failed_s$ be the number of passed and failed test cases that executed program statement s respectively. $Tarantula = (Failed_s \div TotalFailed) \div ((Passed_s \div TotalPassed) + (Failed_s \div TotalFailed))$ and $Jaccard = Failed_s \div (TotalFailed + Passed_s)$. These benchmark techniques were selected because they are amongst the most widely used SBFL techniques [161]; researchers will be familiar with their level of effectiveness, and thus be able to more accurately gauge the effectiveness of ISBFL from a comparative analysis of the techniques.

Recall that an oracle was introduced in Section 3.2.1.2. A mutant is classified as a coincidentally correct fault if this oracle passed all of the test cases that the mutant was executed with, or is otherwise classified as a standard fault. This oracle was used to classify passed and failed test cases for Tarantula, Ochiai and Jaccard. As such, these techniques cannot provide any fault localisation support for coincidentally correct mutants, and so the comparison of these techniques with ISBFL was limited to standard faults. Since the classification of mutants as standard faults only requires some test cases to be labelled as failed by this oracle, there may be some passed test cases, and some of these passed test cases may be labelled as such because of coincidental correctness. In other words, coincidental correctness might affect standard faults. Thus, a comparison based on these faults can still be useful for drawing conclusions about the effectiveness of these techniques and ISBFL on

coincidental correctness.

6.2.8 LOC and Logging

Our fault localisation analysis only considers a subset of the LOC in each subject program. For example, consider the Genetic Algorithm subject program. Recall from Section 3.2.1.2 that we applied mutation testing to 9 classes. These 9 classes consist of 947 of the subject program's LOC (this was calculated based on GAUninstru, which was introduced in Section 3.2.1.1). Our fault localisation analysis is restricted to these 947 LOC, for this subject program. Similarly, only the subject program's LOC that reside in the classes that we applied mutation testing to, in the other subject programs, were included in the fault localisation analysis of these subject programs. In particular, 137, 36, and 60 LOC were considered for Dijkstra's Algorithm, Bubble Sort and Knuth-Morris-Pratt respectively (these were computed based on DAUninstru, BSUninstru, and KMPUninstru respectively (see Section 3.2.3.1)). In other words, our fault localisation analysis was restricted to classes that were relevant to our experiments.

During the execution of a test case tc_i, a logging tool is used to produce a log file $ExecutionTrace_i$, which is a record of the program statements that executed during the execution of tc_i. One feature of the logging tool that was used in our experiments is that it does not log the execution of import statements, package declarations, exceptions and statements that simply mark blocks of code e.g. "{", "}", or "catch(Exception e){}". Additionally, a small number of redundant lines of code exist in DAUninstru, BSUninstru, and KMPUninstru, that were removed from the Dijkstra's Algorithm, Bubble Sort and Knuth-Morris-Pratt subject programs; consequently the logging tool does not log the execution of these lines. Thus, the logging tool is restricted to a subset of the aforementioned 947, 137, 36, and 60 LOC in the Genetic Algorithm, Dijkstra's Algorithm, Bubble Sort, and Knuth-Morris-Pratt subject programs respectively. The same logging tool and LOC restrictions are used for ISBFL, and the other three benchmark techniques detailed above. This ensures fairer comparisons between the techniques i.e. the differences in the effectiveness of the techniques cannot be explained by the logging function.

6.3 Results and Discussion

Recall that 30 mutants were generated for the experiment that was designed to address RQ1. Interlocutory Testing was applied to these mutants; 29/30 were killed. Section 6.3.1 explores ISBFL's feasibility using these 29 mutants. Similarly, we exposed the 60 mutants, that were generated to answer RQ2 — RQ4, to Interlocutory Testing. 47/60 of these mutants were killed. Several experiments were conducted, in which ISBFL was applied to these 47 mutants; Sections 6.3.2 to 6.3.4 present the results of these experiments.

6.3.1 RQ1. Is ISBFL a feasible debugging technique?

To determine whether ISBFL is a feasible debugging technique, we used 100 test cases to exercise ISBFL on 29 mutants that were derived from Dijkstra's Algorithm, Bubble Sort, and Knuth-Morris-Pratt.

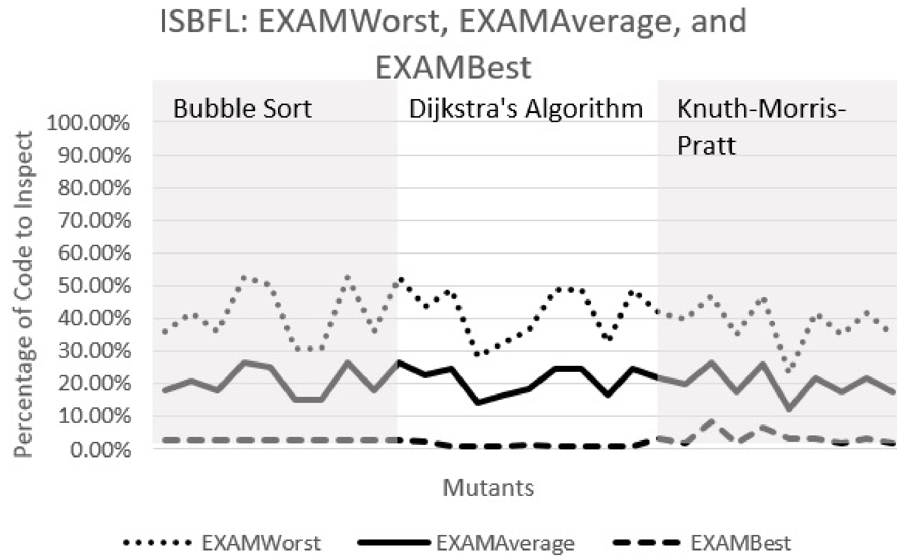

Figure 6.2: ISBFL's EXAMBest, EXAMAverage, and EXAMWorst measures for Bubble Sort, Dijkstra's Algorithm and Knuth-Morris-Pratt

Figure 6.2 shows the percentage of code that one must inspect to locate a fault in the SUT, with the assistance of ISBFL. Different areas of the graph correspond to different subject programs. The graph indicates that ISBFL can offer some degree of fault localisation support for all of the programs. This demonstrates that the technique is feasible.

The graph also indicates that ISBFL obtained a comparable level of performance across each of the subject programs. This suggests that the technique can offer a similar level of fault localisation support for different subject programs. Interestingly, one can also observe that the level of fault localisation support offered by ISBFL can vary for different mutants. This phenomenon will be explored in Section 6.3.3.

6.3.2 RQ2. How effective is ISBFL at localising faults?

This section and Section 6.3.3 use the Genetic Algorithm subject program, 100 test cases, and 47 mutants to address RQ2 and RQ3 respectively.

Figure 6.3 shows the percentage of code that must be inspected to find the faulty line in each mutant, with the assistance of ISBFL, as measured by EXAMWorst, EXAMAverage, and EXAMBest. The graph illustrates that ISBFL's fault localisation effectiveness varies substantially for different mutants. We applied K-Means Clustering to EXAMAverage and identified three distinct clusters of mutants. These clusters are highlighted in the graph. We compared these clusters, in terms of their EXAMAverage, using the Kruskal Wallis H test[2], and found that the difference was statistically significant ($p < 0.05$).

Table 6.1 presents descriptive statistics of each cluster, based on their EXAMAverage. It shows that ISBFL can substantially reduce the number of LOC that must be manually inspected to find a fault. Interestingly, the table also shows that each cluster is tight. This means that the fault localisation effectiveness of ISBFL is similar for mutants that are members of the same cluster. Given that the

[2]The Kruskal Wallis H test is a test statistic that can be used to compare three or more groups, based on a continuous variable [151]. This test statistic is non-parametric, and so can be used in situations in which a parametric statistic's assumptions have not been satisfied [151].

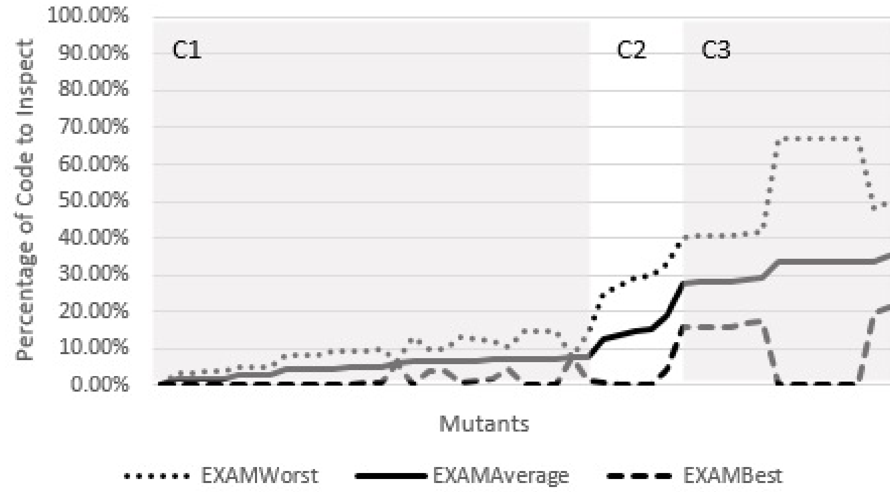

Figure 6.3: ISBFL's EXAMBest, EXAMAverage, and EXAMWorst measures

	Cluster 1 EXAMAverage	**Cluster 2 EXAMAverage**	**Cluster 3 EXAMAverage**
Minimum	0.05%	12.72%	27.82%
Mean	4.87%	14.93%	31.44%
Maximum	7.76%	18.80%	35.16%
Count	28	5	14
Standard Deviation	0.022293	0.020847	0.026816

Table 6.1: Descriptive statistics of each cluster, based on EXAMAverage

fault localisation effectiveness of ISBFL is significantly different for mutants in different clusters, this suggests that the fault localisation effectiveness of ISBFL may be dictated by fault type.

Let $MoreSus$ be the number of LOC that are more suspicious than the faulty line. Recall that $EXAMBest = MoreSus + 1$. Thus, EXAMBest is a very close approximation of $MoreSus$ (i.e. it only deviates by +0.11% in this experiment). As such, it can be used to gauge the percentage of LOC that must invariably be inspected first, before the faulty line has a chance of being inspected. Figure 6.3 illustrates the EXAMBest measures of each mutant. 26 mutants have an EXAMBest of 0.11%, and 7, 6, and 8 mutants have EXAMBests ranging from 0.32 to 1.69%, 3.70 to 7.50%, and 15.95 to 20.91% respectively. EXAMBest varies substantially for different mutants. However, most mutants obtain a low EXAMBest, which means very few LOC were deemed to be more suspicious than the faulty line in most cases.

We define EXAMDiff to be the difference between EXAMWorst and EXAMBest ($EXAMDiff = EXAMWorst - EXAMBest$). EXAMDiff represents the percentage of LOC that have the same suspiciousness score as the faulty line. Low measures of EXAMDiff means that there is a higher probability that the faulty line will be found sooner, when one is inspecting the set of LOC represented by EXAMDiff. Figure 6.3 shows that EXAMDiff's between 0 to 4.96%, 6.02 to 14.36%, and 23.86 to 29.67%, and an EXAMDiff of 66.84% were obtained by 10, 18, 13 and 6 mutants respectively. The variations in EXAMDiff are extreme, but tend to skew more towards lower measures.

The graph also shows that the majority of mutants obtain a low EXAMBest and EXAMDiff. One can also observe that the mutants that obtain the highest measures of EXAMBest don't have the highest EXAMDiff, and that the mutants that obtain the highest measures of EXAMDiff have low measures of EXAMBest.

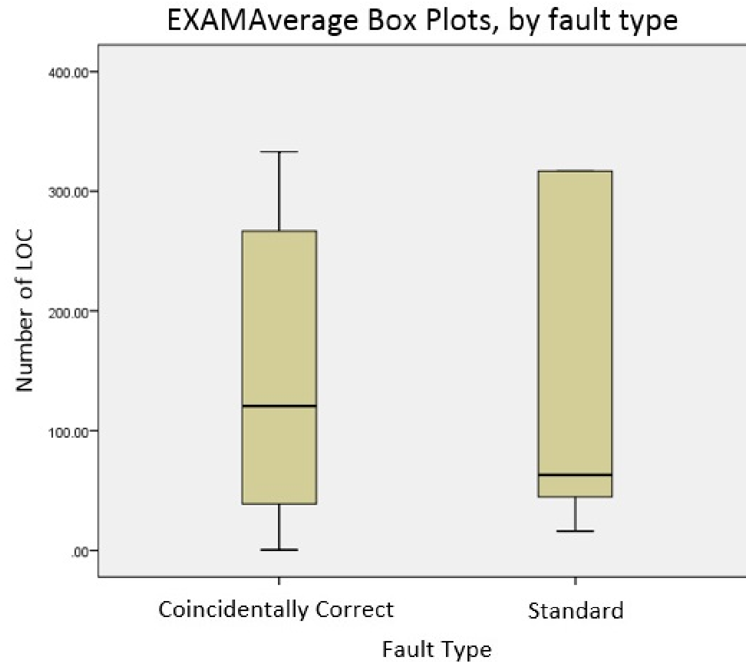

Figure 6.4: Box Plots of the EXAMAverage of standard and coincidentally correct faults

Figure 6.4 presents two box plots that summarise the descriptive statistics of the EXAMAverage for standard and coincidentally correct faults. The box plots show that ISBFL obtains a comparable level of effectiveness for standard and coincidentally correct faults. The difference is not significant (Mann-Whitney U: $p > 0.05$). This suggests that the technique can operate to a similar degree of effectiveness when coincidental correctness is either present or absent. This indicates that the findings in this section e.g. the substantial reductions in the number of LOC that must be manually inspected, are applicable to both standard and coincidentally correct faults.

6.3.3 RQ3. What are the factors that affect the fault localisation effectiveness of ISBFL?

Section 6.3.2 revealed that the fault localisation effectiveness of ISBFL varied for different mutants. This section presents a series of observations that were made, that may explain this variation.

6.3.3.1 Observation One: PISBFL

Our first observation is that all 6 of the mutants that were handled by PISBFL were grouped in the same cluster — Cluster 3. Since ISBFL performed worse for mutants in this cluster than for mutants in other clusters, this suggests that DISBFL is more effective than PISBFL. To confirm this, we computed the average EXAMAverage of DISBFL and PISBFL, which were 10.98% and 33.47% respectively, and compared the EXAMAverage of DISBFL and PISBFL with the Mann-Whitney U test. The difference was statistically significant.

We inspected the 6 mutants that were handled by PISBFL. We found that all 6 of these mutants

were only detected by 1 probabilistic IR. Thus, in all 6 cases, all of the LOC that were deemed to be suspicious by probabilistic IRs, were awarded an equal suspiciousness score. Additionally for each mutant, these probabilistic IRs were evaluated after a large number of LOC had been executed, so a large number of program statements were deemed to be suspicious by these IRs. This explains why EXAMAverage was particularly high for these mutants. However, PISBFL is only used when DISBFL fails to provide any fault localisation support. Thus, despite PISBFL's comparatively poor performance, it did add value.

These 6 mutants are also the mutants in Figure 6.3 that have an extremely high EXAMDiff, with a very low EXAMBest. The discussion above also explains this phenomenon.

6.3.3.2 Observation Two: Frequencies of Constant and Transient Program Statements

The discussions in this section are limited to mutants that were detected by DISBFL. We call a program statement a "constant program statement", if it executes in all test cases. Conversely, we define a "transient program statement" to be a program statement that executes in some test cases, but not others.

We observed that the average EXAMAverage of mutants in which the faulty LOC was a transient program statement was 4.63%, and the average EXAMAverage of mutants in which the faulty LOC was a constant program statement was 15.96%, and that the difference was statistically significant (Mann-Whitney U: $p < 0.05$). This suggests that the nature of the fault is an important determinant of fault localisation effectiveness. The faulty program statement in each mutant in clusters 2 and 3 was a constant program statement. All of the mutants in which the faulty program statement was a transient program statement are in cluster 1. This explains why cluster 1 has a lower EXAMAverage than clusters 2 and 3.

Interestingly, the faulty line in 10/28 of the mutants in cluster 1 were also constant program statements, and yet these mutants obtained a significantly lower EXAMAverage than mutants in clusters 2 (Mann-Whitney U: $p < 0.05$) and 3 (Mann-Whitney U: $p < 0.05$). We investigated this phenomenon. We found that in the mutants that are members of clusters 2 and 3, a large number of constant program statements were executed before the IR instance that detected the fault had been evaluated. This meant that these faulty program statements were deemed to be equally suspicious to a large number of constant program statements; thus these mutants were skewed towards higher measures of EXAMDiff. The difference in EXAMDiff between clusters 2 and 3 is not significant (Mann-Whitney U: $p > 0.05$). The IRs that detected the 10 aforementioned mutants in cluster 1, were evaluated after fewer constant program statements were executed. This meant that these 10 mutants obtained a lower EXAMDiff than the mutants in clusters 2 and 3. A Mann-Whitney U test demonstrated that this was significant ($p < 0.05$). This may explain why the EXAMAverage of mutants in cluster 1 was lower than mutants in clusters 2 and 3. These findings suggest that the point in the execution trace that an IR is evaluated at is an important determinant of fault localisation effectiveness.

The discussion above exposes why mutants in cluster 1 obtained a lower EXAMAverage than mutants in clusters 2 and 3, but does not explain why mutants in cluster 2 achieved a lower EXAMAverage than mutants in cluster 3. We therefore decided to investigate this. We found that the faults in these mutants were only detected by IRs that were evaluated after a large number of transient

program statements had an opportunity to execute. The number of transient program statements that were executed in mutants in clusters 2 and 3 were comparable (Mann-Whitney U: $p > 0.05$). Unlike the faulty program statements in these mutants, these transient program statements may not have executed in all passed test cases, or in all failed test cases before the failed IR was evaluated, which means that they can appear in a different proportion of passed program spectra, and a different proportion of failed program spectra, and therefore can be either awarded a higher or lower suspiciousness score than the faulty statement, depending on these proportions. In the case of cluster 3, a large number of transient program statements obtained a higher suspiciousness score than the faulty program statement. This explains why these mutants are skewed towards higher EXAMBest's. Far fewer transient program statements in mutants in cluster 2 obtained a higher suspiciousness score than the faulty statement. Thus, cluster 2 obtained a significantly lower EXAMBest than cluster 3 (Mann-Whitney U: $p < 0.05$); this explains why mutants in cluster 2 have a significantly lower EXAMAverage than mutants in cluster 3. This suggests that the control flow of the program is an important determinant of fault localisation effectiveness, since control flow determines which program statements are executed in each test case. It also indicates that the fault, IR, and test data are important, since these determine which test cases are passed and failed test cases.

Finally, we also observed that in 2/10 of the aforementioned mutants in cluster 1, significantly fewer transient program statements had the opportunity to execute before the fault had been detected by IRs, in comparison to clusters 2 and 3 (Mann-Whitney U: $p < 0.05$). This meant that the total number of program statements that had the scope to be deemed more suspicious than the faulty line was smaller for these mutants. This explains why these mutants obtain a low EXAMBest. The comparatively larger number of transient program statements in the other 8/10 mutants were largely deemed to be less suspicious than the faulty line. The EXAMBest of cluster 1 is comparable to the EXAMBest of cluster 2 (Mann-Whitney U: $p > 0.05$).

6.3.3.3 Observation Three: IRs

Figure 6.5 shows the minimum, average and maximum EXAMAverage of each IR(s). The graph clearly indicates that average EXAMAverage varies considerably, depending on which IR(s) detected the fault. This suggests that one's choice of IRs can have a substantial impact on fault localisation effectiveness.

The graph illustrates that most of the IRs had a very similar minimum, average and maximum EXAMAverage. This indicates that the performance of most IRs is reasonably consistent. However, TerminateGA and TournamentComposition varied substantially, in terms of their measures of EXAMAverage. We decided to investigate this. The faulty LOC in each of the mutants that had been handled by TerminateGA were constant program statements, and were therefore equally suspicious to all other constant program statements. We found that the number of constant program statements that executed in these mutants varied substantially because of the way in which each of these mutants modified the SUT's control flow. This supports our previous observation that the nature of the fault is an important determinant of fault localisation effectiveness. With regards to TournamentComposition; the difference in performance for different mutants can be explained by the number of transient program statements that were deemed to be more suspicious than the faulty line — see Section 6.3.3.2.

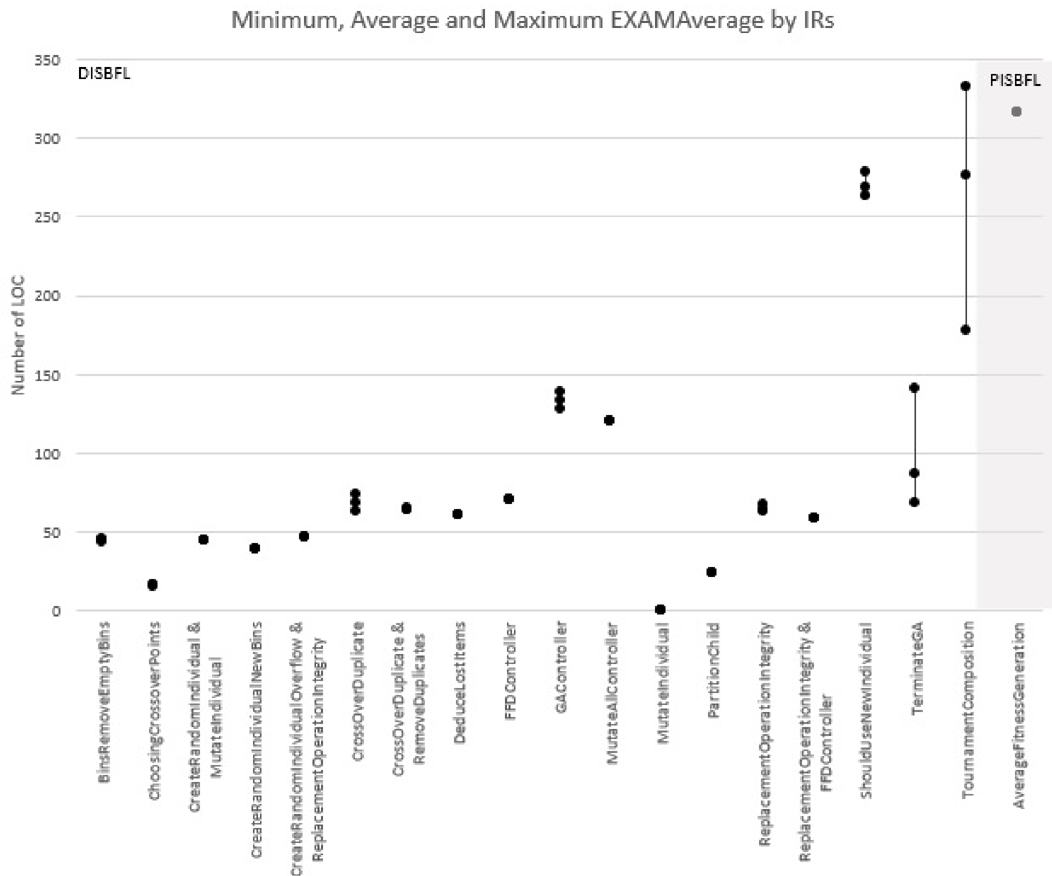

Figure 6.5: Minimum, Average, and Maximum EXAMAverage of each IR(s)

6.3.3.4 Observation Four: Fault Location

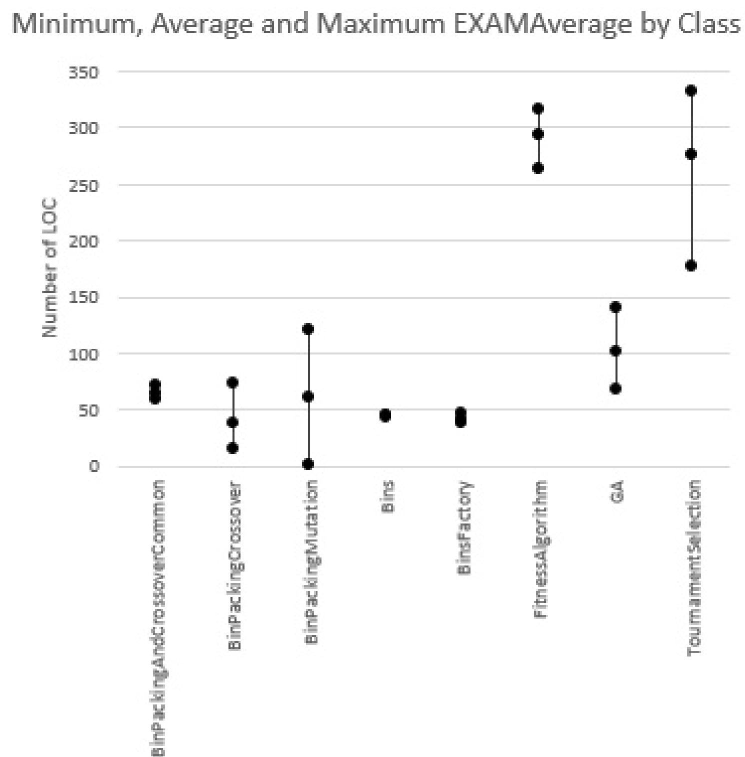

Figure 6.6: Minimum, Average and Maximum EXAMAverage by class

Figure 6.6 shows the minimum, average and maximum EXAMAverage obtained by ISBFL for

faults in different Java classes. We can observe that the average EXAMAverage of ISBFL varies for faults in different classes. This suggests that the location of the fault can have an impact on ISBFL's fault localisation effectiveness.

The Bins and BinsFactory classes have the lowest maximum EXAMAverage and amongst the lowest average EXAMAverage. The LOC in these classes are involved in the initialisation of the algorithm. This meant that faults in these classes manifested early in some test cases. These faults were also detected by IRs, early in the execution trace. This explains why ISBFL's fault localisation effectiveness for faults in these classes was particularly high. Interestingly, the LOC in the FitnessAlgorithm class are also used in the initialisation phase, but faults in this class were skewed towards higher measures of EXAMAverage. An investigation revealed that faults in this class were invariably detected by IRs that are evaluated at a late point in the execution trace. This supports our findings above, regarding the importance of the location of the IR.

The graph also shows that there is a large amount of variance in terms of the minimum, average and maximum EXAMAverage for mutants in several classes. This indicates that the stability of ISBFL's performance for different classes can vary. It also suggests that other important factors may influence fault localisation effectiveness and are responsible for this variance e.g. as discussed above, the location of the IRs that detected the faults.

6.3.4 RQ4. How does the fault localisation effectiveness of ISBFL compare to other SBFL techniques?

In this section, we compare ISBFL to three well-known SBFL techniques — Tarantula, Ochiai, and Jaccard. Our comparisons are based on the same subject program and test cases, and a subset of the mutants that were used to address RQ2 and RQ3. In particular, our comparisons are based on the fault localisation effectiveness of these techniques, for mutants that all of these techniques can provide fault localisation support for. As discussed in Section 6.2.7, Tarantula, Ochiai, and Jaccard were only applicable to standard faults. Thus, we narrowed our focus to these faults.

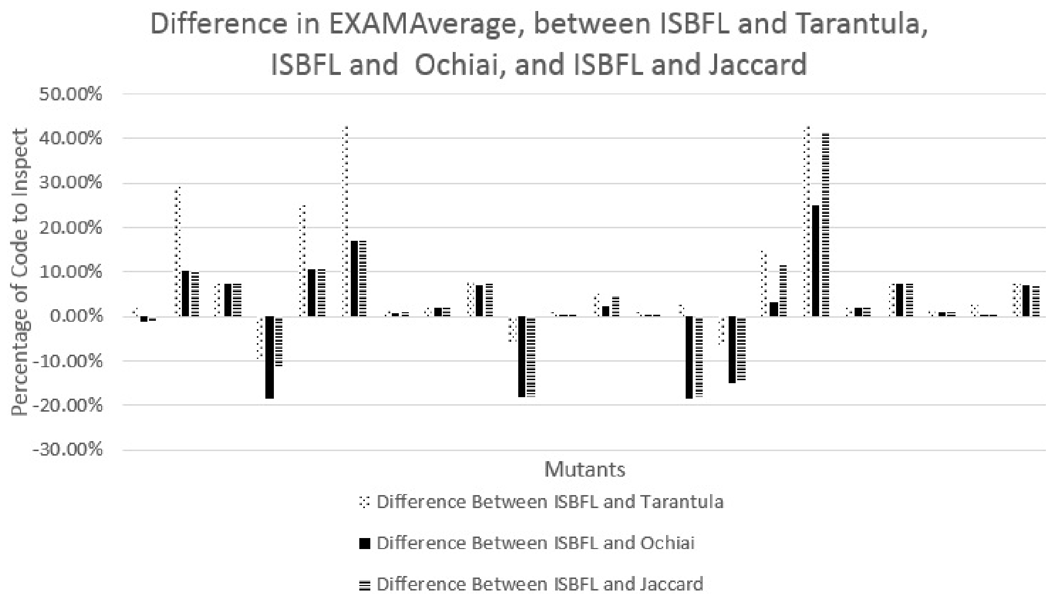

Figure 6.7: Difference in EXAMAverage between ISBFL and the benchmark techniques

Each cluster of bars in Figure 6.7 corresponds to one mutant and each bar in the cluster represents

one of the benchmark techniques. The height of a bar communicates the difference in EXAMAverage between ISBFL and the benchmark technique represented by that bar. For example, the first, second and third bar on the graph shows that Tarantula's EXAMAverage was 1.95% higher than ISBFL's, Ochiai's EXAMAverage was 1.11% lower than ISBFL's, and Jaccard's EXAMAverage was 1% lower than ISBFL's respectively, for the mutant represented by that cluster of bars.

The graph shows that ISBFL outperforms the benchmark techniques for most mutants. The average EXAMAverage of ISBFL, Tarantula, Ochiai and Jaccard are 12.74%, 21.18%, 14.28% and 15.95% respectively. We conducted a series of Mann-Whitney U tests that compared ISBFL to each of the benchmark techniques, and all of the comparisons yielded a statistically significant result. Despite this, the graph also shows that some of the benchmark techniques outperformed ISBFL for five mutants. An investigation of these mutants revealed that they were all handled by PISBFL. PISBFL only handled 6 mutants in total, thus it outperformed the benchmark techniques in one case. This suggests that some of the benchmark techniques are typically more accurate than PISBFL.

This also means that DISBFL outperformed the benchmark techniques in all cases, which indicates that DISBFL may be generally more effective than the benchmark techniques. There are two possible explanations for this. Let *tc* be a test case that was marked as failed by both DISBFL and the benchmark techniques. In DISBFL, the program statements that executed in *tc*, after the failed IR in *tc* was evaluated, were excluded from the program spectra of *tc*. These program statements would not have been excluded from the failed program spectra used by the benchmark techniques; thus there would have been more program statements that had the potential to be deemed to be equally, or more suspicious than the faulty line in the benchmark techniques.

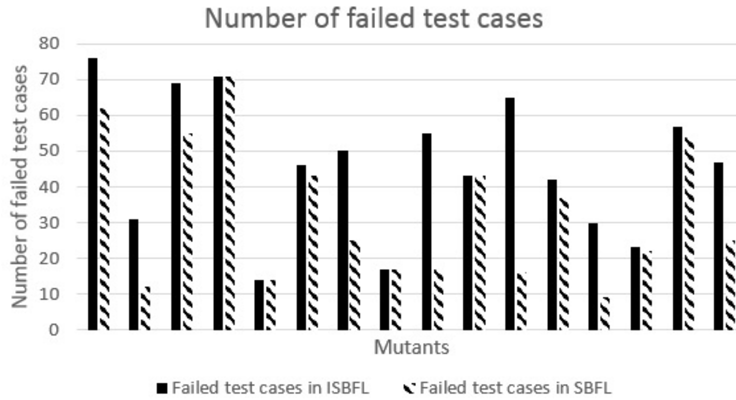

Figure 6.8: Number of test cases that are classified as failed by DISBFL and SBFL.

Secondly, coincidental correctness may have caused some of the failed test cases to be misinterpreted as passed test cases by the three benchmark techniques, and may not have affected the classification of these test cases for DISBFL. We investigated this possibility. Figure 6.8 shows the total number of test cases that were marked as failed test cases by DISBFL and the three benchmark techniques. The difference between the three benchmark techniques in our experiment is solely the suspiciousness score metrics that were used; thus the same test cases were deemed to be passed and failed by the benchmark techniques. To that end, we represented all three techniques in Figure 6.8 with the "SBFL" bar. The graph clearly shows that DISBFL always classifies at least as many test cases as failed as SBFL, and classifies more test cases as failed very often. The test cases that are classified as failed by DISBFL and passed by SBFL are coincidentally correct. This suggests that

the second possibility was realised, and indicates that ISBFL can reduce the impact of coincidental correctness on SBFL techniques.

6.3.5 Discussion

Interestingly, the overall fault localisation effectiveness obtained by ISBFL for the subject programs that were used to answer RQ1, was 20.63%, which is comparable to the overall fault localisation effectiveness that was observed for the subject program that was used to explore RQ2 — RQ4, which was 13.86%. This supports our findings in Section 6.3.1.

However, a comparison of Figures 6.2 and 6.3, revealed that the nature of the fault localisation support offered to these subject programs varied substantially. In particular, ISBFL offers a more consistent level of fault localisation support for the three subject programs that were used for RQ1, than for RQ2 — 4.

The control flow of a system determines whether a program statement can be a transient program statement. To illustrate, consider a program statement p. If p is the first LOC in the system, then it must be invariably executed in all test cases and so cannot be transient. Alternatively, suppose that p is in the body of an if statement; p can be transient because the conditional may allow it to execute in some test cases, but prevent it from executing in others. Since each of the subject programs have a different control flow structure, they would also have had a different number of transient and constant program statements. This disparity may explain the differences in results.

As was discussed in Section 6.3.3.3, different IRs offer different levels of fault localisation support. Different IRs were used in each of the experiments. This might also explain the difference in results. Section 6.3.3.4 revealed that the location of the fault has an important impact on the effectiveness of fault localisation. This could be another factor that could explain the difference in results.

The discussion in this section suggests that the nature of the fault localisation support offered by ISBFL may vary for different systems. Interestingly, with the exception of different IRs being used across different experiments, all of the factors that could be responsible for this variance could also affect standard fault localisation techniques.

6.4 Threats to Validity

The threats to validity that were discussed in Section 3.5 are relevant to the experiments conducted in this chapter. There were also several additional threats to validity that affected the experiments in this chapter, and some of the threats that were discussed in Section 3.5 warrant further, context specific discussions. This section outlines these additional threats and presents these discussions.

6.4.1 Internal Validity

The Genetic Algorithm subject program was non-deterministic. Let tc be a test case for this subject program. Suppose that tc was executed twice, and produced two execution traces, et_1 and et_2. Because of non-determinism in the system, $et_1 \neq et_2$. Thus, if the effectiveness of ISBFL and a benchmark technique was measured based on et_1 and et_2 respectively, the differences in the effectiveness of these two techniques might be partly explained by the differences in et_1 and et_2. To mitigate this as

a confounding factor for the comparison, we executed each test case tc_i once to produce a single corresponding execution trace et_i. ISBFL and all of the benchmark techniques were all applied to et_i.

Recall from Section 3.5 that our logging tool can increase code coverage. Since ISBFL and the benchmark techniques used the same execution trace (that was produced by this logging tool), they would have all been affected by this. Other logging tools may not extend a test cases coverage, and thus ISBFL and these benchmarks may achieve a higher level of effectiveness with these logging tools. Thus a threat to repeatability might be one's choice of logging tool.

Recall from Section 6.2.8 that our logging tool can only record the execution of a subset of the lines of code of the SUT. In order to configure our logging tool to be able to record the execution of these lines of code, it was necessary to manually annotate every one of these lines of code. Such a task has scope for errors e.g. we may have accidentally annotated a logging function line of code. However, the approach that we used to systematically identify lines of code that should be annotated was similar to the approach that was used in Section 3.2.1.1 to systematically identify lines of code that should contribute towards an estimate of the subject program's size. Thus, the number of annotation errors is likely to be low, and thus the impact on the results would have been negligible.

Let $IRInstance_a = \langle IR_y, States_a \rangle$ be an IRInstance i.e. execution trace data was extracted from all of the program states in $States_a$ to evaluate IR_y. Also let $LastState_a$ be a program state in $States_a$, such that $LastState_a$ manifests at a later point in the execution trace than all other program states in $States_a$. In Section 6.1.1 we explained that in ISBFL, $IRInstance_a$ is associated with the distinct set of program statements that executed before $LastState_a$ manifested. In our implementation of the technique, $IRInstance_a$ is instead associated with all of the program statements that executed before our logging tool decided to commit $LastState_a$ to the log file. This decision could be made after program statements that are preceded by $LastState_a$ have executed. This could reduce the effectiveness of ISBFL, and thus cause our results to underestimate its effectiveness.

6.4.2 External Validity

We used four subject programs to evaluate ISBFL. These subject programs varied in terms of size, problem domain, and susceptibility to coincidental correctness. Our evaluation revealed that the overall effectiveness of ISBFL was comparable across this diverse range of systems. However, it also revealed that the nature of the fault localisation support that can be offered by ISBFL can vary for different subject programs. These observations provide us with some indication about the generalisability of the results. However, we recognise that four subject programs is not a sufficiently large sample to make absolute claims about the overall generalisability of the technique. We would therefore like to address this in future work.

6.4.3 Construct Validity

The EXAM Score is a measure of the percentage of LOC that must be checked before the first faulty program statement has been found [191]. Let *EquallySus* be the set of LOC in the SUT that have the same suspiciousness score as the faulty line. Wong and Debroy [191] explain that (in addition to checking all of the LOC that are more suspicious than the faulty program statement) the user may only have to check one line in *EquallySus* in the best case, and all of lines in *EquallySus* in the worst case. They state that there are two variants of the EXAM Score — one that assumes the

best case, and the other which assumes the worst case. Our EXAMBest and EXAMWorst metrics correspond to these measures. These measures have been used by many researchers in the debugging community. In this chapter, we also used a metric called EXAMAverage, which assumes that one must check approximately half of the LOC in *EquallySus*, and is intended to represent the typical case. This metric is correlated with the two aforementioned metrics.

6.5 Conclusions

The fault localisation effectiveness of SBFL techniques can be compromised by coincidental correctness. Interlocutory Testing is a testing technique that was devised for testing systems that are prone to experiencing the effects of coincidental correctness. In this chapter, we introduced a new variant of SBFL called ISBFL, that is the amalgamation of SBFL and Interlocutory Testing. The primary goal of ISBFL was to alleviate the impact of coincidental correctness on SBFL.

We confirmed the feasibility of the technique and investigated its effectiveness. Our investigation revealed that the approach could substantially reduce the number of LOC that must be manually inspected by the tester, despite the presence of coincidental correctness. Thus, we have addressed Objective 4 (see Section 1.1). We also found that the fault localisation effectiveness of the technique varied significantly, depending on which mutants it was applied to. We explored and confirmed several sources of this variation. For example, the location of the IR, fault, and the control flow of the SUT were important factors that determined the fault localisation effectiveness of the technique.

We finally performed a comparative analysis between ISBFL and three widely used benchmark techniques — Tarantula, Ochiai, and Jaccard. The results of these comparisons indicated that ISBFL substantially outperformed these techniques in most cases, because of its capability to account for coincidental correctness. We observed that the DISBFL variant of ISBFL was largely responsible for ISBFL's comparatively better performance; DISBFL consistently outperformed the benchmark techniques. However, the effectiveness of the PISBFL variant was lower than these benchmark techniques in the majority of cases. This indicates that future work that explores alternative, more effective, strategies for conducting ISBFL with Probabilistic IRs might be beneficial.

Another avenue of future work that we would like to explore includes extending ISBFL to incorporate program slicing. Program slicing is a program dependency analysis technique that can compute dependencies between program statements [73]. Such a technique could be used by ISBFL to further refine failed program spectra. In particular, instead of including all of the program statements *AllStmts* that executed before the failed IR was evaluated in a failed program spectra, program slicing could be used to further filter *AllStmts*, by removing program statements in *AllStmts* that did not affect the evaluation of the IR.

In this chapter, we only applied ISBFL to four subject programs. Although the experiments that were based on these subject programs provide some insights into the generalisability of the technique, it does not fully confirm it. Thus, we believe that future work that explores the effectiveness of the technique on more subject programs would be beneficial.

IRs are central to ISBFL. ISBFL leveraged 18 IRs in this experiment. In some situations, the user may find the notion of defining such a large number of IRs unacceptable. As was discussed in Section 3.3.6.2, the process of defining IRs can be partially automated. For future work, we would like

to develop a means of increasing this degree of automation further, since this may alleviate the issue. In the meantime, we hope that the cost of developing IRs for fault localisation might be justified by the fact that they could also be used for testing.

To summarise, this chapter introduced ISBFL, a modified version of SBFL that incorporates Interlocutory Testing for the purpose of mitigating the impact of coincidental correctness on SBFL; this chapter satisfied Objective 4. This chapter in conjunction with the previous chapters have collectively satisfied all of the objectives of the thesis. The next chapter will finally conclude the thesis.

Chapter 7

Conclusions

Many testing and debugging techniques assume that corrupt program states will propagate to the output, and can therefore be detected by inspecting the output. Coincidental correctness is a phenomenon in which a fault corrupts a program state, and this state does not propagate to the output. Thus, coincidental correctness violates this assumption. The violation of this assumption can compromise the effectiveness of such testing and debugging techniques e.g. Metamorphic Testing and Spectrum-based Fault Localisation respectively. The ubiquity of coincidental correctness has been empirically demonstrated by several researchers (see Section 2.8), motivating the research that was described in this thesis. In particular, our research culminated in four techniques that can reduce the impact of coincidental correctness on some of these testing and debugging techniques.

Section 7.1 presents the main contributions of the thesis, and Section 7.2 outlines and discusses the main limitations of our techniques and research, and highlights possible future research directions. Finally, Section 7.3 presents a brief summary of the thesis.

7.1 Contributions

This section outlines the main contributions of the thesis. We conducted a mapping study on the oracle problem; Chapter 2 describes this mapping study. A summary of the main contributions pertaining to the mapping study can be found in Section 7.1.1. The mapping study revealed that coincidental correctness can limit the effectiveness of testing and debugging techniques that assume that corrupt program states will propagate to the output. This inspired the aim of the thesis — to reduce the impact of coincidental correctness on these techniques.

To address the aim of the thesis, we fulfilled the following objectives: Develop a new testing technique that can operate effectively in the presence of coincidental correctness (Objective 1), modify Metamorphic Testing to reduce its susceptibility to coincidental correctness (Objective 2), develop a partial solution to the Equivalent Mutant Problem that can tolerate coincidental correctness and non-determinism (Objective 3), and modify Spectrum-based Fault Localisation, to mitigate the impact of coincidental correctness (Objective 4). These objectives were tackled in Chapters 3, 4, 5, and 6 respectively. A summary of the main contributions that were made in these chapters can be found in Sections 7.1.2 to 7.1.5 respectively.

7.1.1 Mapping Study

The mapping study surveyed research on automated testing techniques that can detect functional software faults in non-testable systems. Discussions about each technique were presented, along with a comparison of these techniques. The material enabled the identification of research opportunities. The mitigation of coincidental correctness from various testing and debugging techniques are examples of such research opportunities. We actively perused some of these research opportunities in this thesis — see Sections 7.1.2 to 7.1.5.

7.1.2 Interlocutory Testing

Coincidental correctness occurs when the output produced by the SUT is plausible, but the execution trace behaviours that should have been responsible for the production of this output were not. We observed that one can predict aspects of the execution trace, based on the nature of the relationship between the input and output. Comparing these predictions against the execution trace is akin to checking whether the output manifested as a consequence of these predicted behaviours; thus, such comparisons are tantamount to directly testing for coincidental correctness. We developed a testing technique that performs such comparisons, called Interlocutory Testing.

Interlocutory Testing has two types of oracles — Deterministic IRs and Probabilistic IRs. Deterministic IRs are applied to aspects of the system that behave deterministically, whilst probabilistic IRs are applied to non-deterministic aspects of the system. Unfortunately, we realised that Probabilistic IRs were susceptible to false positives. To that end, we developed a statistics-based evaluation method that could be adopted by Probabilistic IRs, to reduce the incidence of false positives. This evaluation method was empirically demonstrated to be an effective means of reducing false positives.

48, 4, 1, 1 and 3 IRs were developed for the Genetic Algorithm, Dijkstra's Algorithm, Bubble Sort, Binary Search and Knuth-Morris-Pratt programs respectively. Three experiments were conducted based on these IRs and programs. These experiments indicated that Interlocutory Testing is a feasible testing technique, can be effective in the presence or absence of coincidental correctness with a relatively small number of IRs, was capable of operating in a wide range of systems e.g. systems that vary in terms of their susceptibility to the oracle problem, coincidental correctness, and degrees of non-determinism/determinism, and that one's choice of test suite has little impact on the overall effectiveness of the technique. This means that Interlocutory Testing satisfied Objective 1.

The experiments also indicated that Deterministic IRs can be more effective than Probabilistic IRs, context specific IRs might be more effective than general IRs, IRs that mainly employ the IOR-ID-Detection strategy could be more effective than IRs that mostly use the IOR-Only-Detection strategy, and that IRs that emphasise testing highly coupled areas of the SUT have the potential to be more effective than IRs that emphasise testing other areas of the SUT. We also found that it could be beneficial to leverage multiple IRs that can detect the same faults, and that one's choice of test suite can be more important for certain types of faults than for others.

Finally, a comparative analysis of Interlocutory Testing and traditional testing techniques, in terms of effectiveness and usability, was presented.

7.1.3 Interlocutory Metamorphic Testing

It has been reported that the effectiveness of Metamorphic Testing can be negatively affected by coincidental correctness. Our empirical results in Section 4.3.2 supported these reports.

This motivated us to extend Metamorphic Testing with Interlocutory Testing, to alleviate the impact of coincidental correctness on Metamorphic Testing. This extended version of Metamorphic Testing is called Interlocutory Metamorphic Testing (IMT). An oracle in IMT is called an IMR. An IMR consists of one MR and a set of IRs. In IMT, an IMR's MR is first evaluated multiple times to obtain a set of execution traces. The IMR's IRs are then evaluated based on these execution traces. If the MR or any of these IRs fail, then the IMR reports a failure.

Nine IMRs were developed for the five programs listed in Section 7.1.2. We conducted a series of experiments based on these IMRs and programs. Some of the experiments explored the feasibility and effectiveness of IMT. The results indicated that IMT is feasible for a diverse range of programs. They also demonstrated that IMT could improve the effectiveness of Metamorphic Testing for both coincidentally correct and standard faults. These findings demonstrate that we have completed Objective 2.

Another experiment investigated the differences between Interlocutory Testing and IMT. This experiment revealed that IMT might have access to fewer IRs than Interlocutory Testing, and that this means that Interlocutory Testing can detect a wider range of faults. It also indicated that IMT has a higher FDR, which means that IMT is more likely to detect a fault. The techniques found different faults, which suggests that IMT is a useful complementary technique. This investigation also exposed some additional insights about Interlocutory Testing e.g. different IRs might have different FDRs. It also unearthed some insights about IMT. For example, an IMR's coverage in terms of the execution trace is important — not just the source code.

The final experiment explored the impact of the test suite on IMT. The results demonstrated that IMT's performance for one test suite was not significantly different than for another.

7.1.4 Interlocutory Mutation Testing

Techniques have been developed to alleviate the Equivalent Mutant Problem e.g. TEMDT. Such techniques can be hindered by coincidental correctness and non-determinism. This motivated us to develop a new technique that can alleviate this problem under these conditions, called Interlocutory Mutation Testing (IMuT). In IMuT, IRs are developed to make predictions that are based on how the SUT actually operates (instead of how it should operate). Such IRs can be applied to mutants. The failure to satisfy an IR means that the behaviour of the mutant deviated from the SUT's behaviour, and thus indicates that the mutant is non-equivalent. If all of the IRs are satisfied, then the mutant is said to be equivalent.

We conducted a series of experiments that explored the mutant classification accuracy of IMuT, as well as the resultant manual inspection effort reductions. These experiments indicated that IMuT can classify mutants accurately in either the presence or absence of coincidental correctness and/or non-determinism, with a relatively small number of IRs. By implication, this means that Objective 3 has been accomplished. These results were comparable across subject programs. Conversely, the extent to which manual inspection effort was reduced by IMuT varied across subject programs. Some of the reasons for this variance included the fact that different IRs had been used, and that a different

proportion of equivalent to non-equivalent mutants had been generated by MuJava. Despite this, the results indicated that the manual inspection effort reductions were substantial in all cases.

We also performed a comparative study between IMuT and TEMDT. The study revealed that IMuT substantially outperformed TEMDT. Recall that TEMDT operates by comparing (aspects of) the outputs of the SUT and mutant. The study also indicated that the mutant classification accuracy of TEMDT might be dependent on the aspects of the output it bases its comparisons on.

7.1.5 Interlocutory Spectrum-based Fault Localisation

Coincidental correctness can reduce the effectiveness of Spectrum-based Fault Localisation (SBFL) techniques. To that end, we combined SBFL with Interlocutory Testing, to form Interlocutory Spectrum-based Fault Localisation (ISBFL). ISBFL uses IRs to classify test cases as passed or failed, and refine program spectra. The precise method (i.e. either DISBFL or PISBFL) that is used to achieve this depends on the IR type.

Two experiments were conducted and demonstrated that ISBFL is feasible, and can offer substantial reductions in the number of LOC that one must inspect to find the faulty LOC, despite the presence of coincidental correctness. ISBFL therefore satisfies Objective 4. We explored factors that affect the effectiveness of ISBFL. Some of the most prominent factors included: whether DISBFL or PISBFL was used, the nature and location of the fault, the choice of IRs and their location, test data, control flow, and the frequency of constant and transient program statements across passed and failed test cases.

The results also indicated that ISBFL offered a similar level of fault localisation support for different subject programs, but the nature of this support differed. Potential reasons for this include the aforementioned factors that affect the effectiveness of ISBFL. Finally, we also compared ISBFL to three well-known benchmark techniques — Tarantula, Ochiai, and Jaccard. The results indicated that ISBFL is more effective than these techniques because of its capability to reduce the impact of coincidental correctness.

7.2 Limitations and Future Work

In this section we outline the main limitations of our techniques and research, and highlight potential areas of future work.

7.2.1 Limitations of the Techniques

The main limitations of our techniques revolve around usability. This section explores these limitations. One issue concerns the amount of effort that is required to construct an IR. The IR identification and development process is currently highly manual, and requires the user to have in-depth knowledge about the problem domain and the SUT's implementation details. Thus, the technique can be expensive and/or difficult to apply. Although several tools and techniques exist that might partially automate the IR construction process and thus alleviate these issues e.g. program slicing tools like Indus [159] or WALA [82], and invariant detection tools like Daikon [74] (see Section 3.3.6.2), we believe that there is scope for further automation. Thus, we would like to explore this possibility in future work.

Secondly, the evaluation method that is used with PIRs to curtail false positives can be difficult to apply. This evaluation method requires one to be aware of the typical false positive rate of each Probabilistic IR in terms of a test case, and an entire test suite; determining these typical false positive rates can be difficult. As discussed in Section 3.1.2.2 these values can be extrapolated from empirical data. Such data is easily accessible by IMuT — one can simply generate the data from the SUT. However, such data may not be readily available to the other techniques — one's ability to obtain such data may depend on external factors e.g. availability of a reference implementation. In cases where an RI is not available, one may have to rely on one's own expertise, consult domain experts, or analyse the randomised properties of the SUT. The former two alternatives are ad-hoc and lack systematic guidance, and it is unclear how one might implement the third alternative. We therefore believe that future work that explores different evaluation methods would be beneficial. One such evaluation method was proposed in Section 3.6.

We made four observations that we believe are particularly important for informing the design of effective IRs. In particular: context specific IRs can obtain a greater level of effectiveness than general IRs, IRs that that prioritise the IOR-ID-Detection strategy might be more effective than IRs that largely use the IOR-Only-Detection strategy, the location that an IR emphasises testing is important e.g. IRs that target highly coupled areas of the system tend to perform more effectively, and Deterministic IRs can obtain a greater level of effectiveness than Probabilistic IRs. However, this set of observations does not constitute a comprehensive set of guidelines on how one should design effective IRs. Thus, another issue is the lack of such a comprehensive set of guidelines. This will be explored in future work.

7.2.2 Limitations of the Research

There are two particularly noteworthy limitations of our research. As discussed in Section 7.2.1, our techniques require the user to have an in-depth understanding of the problem domain and the implementation details of the SUT. This affected the total number and type of subject programs that could be feasibly included in our experiments. The first issue in our research is the number of subject programs that were used to evaluate our techniques; throughout our experiments, we only used five subject programs. This limited the strength of the claims that we could make about the generalisability of our techniques. For example, none of the five subject programs had graphical user interfaces (GUI), thus it is unclear how our techniques might perform when applied in this specific context. Another example includes the fact that none of our subject programs are the size of industrial scale systems, and so it is unclear whether our techniques can scale up to such systems. However, we would like to remark that our results do suggest that one could use our techniques for small, critical parts of such systems. Despite this, the five subject programs varied considerably in terms of size, problem domain, and susceptibility to coincidental correctness. The consistency of our results across such a diverse range of subject programs provides some evidence regarding their generalisability. For future work, we would like to investigate the effectiveness of our techniques on more subject programs, to strengthen the evidence base for these claims.

Let R_1 and R_2 denote the results of the main experiment in two different chapters. We postulated that if a noteworthy difference was observed between R_1 and R_2, then some interesting insights about the techniques that were explored in these respective chapters might be revealed by a comparative

analysis between R_1 and R_2. Such an analysis would be susceptible to various confounding factors e.g. mutants, non-determinism, and the test suite. To increase our confidence in the results of such an analysis, we therefore decided to eliminate one of the confounding factors — the test suite. This was the rationale behind using the same test suite for the main experiment in each chapter. The lack of test suite diversity across the main experiments from each chapter is another noteworthy limitation of our research.

This lack of diversity is unlikely to have had a meaningful impact on our results, since the main experiments primarily used the Genetic Algorithm subject program, which generated most of the test data randomly. Thus, different executions of the same test suite would have exposed the IRs to different test data. Our experiments also indicate that increasing the diversity of test suites is unlikely to have had a meaningful impact on the results. For example, Section 3.3.5 demonstrated that effectiveness of the technique was very consistent across 30 different test suites. Additionally, the experiment that was described in Section 4.3.4 compared the effectiveness of our IRs on two different test suites and the difference was not statistically significant. Furthermore, our IRs for the other four subject programs obtained a consistent level of performance across chapters, despite being exposed to different test suites.

7.2.3 Future work

There are a plethora of other potential future research directions that could be investigated. For example, as discussed in Section 6.5, the fault localisation effectiveness of ISBFL may be improved by integrating ISBFL with program slicing. Another example might be an IR centric test adequacy criterion, where a test suite is deemed to have adequate coverage, if each IOR in each IR is evaluated at least once by the test suite. The effectiveness of our techniques may improve if they were used in conjunction with such a test suite.

In this thesis, we only assessed our techniques based on their feasibility and effectiveness, and to some extent their usability. Other important quality attributes remain unexplored. For example, we did not investigate the performance of our techniques, which is an important factor in certain contexts e.g. time sensitive software applications and multi-threaded programs. Other areas of future work may include an investigation of such quality attributes.

Another potential avenue of future work includes replication studies. Throughout this thesis, we have made information available, that is necessary to enable the replication of our work. To illustrate, Sections 3.2, 3.5, 4.2, 4.4, 5.2, 5.5, 6.2, and 6.4, and Appendices A, C, D, E, and F provide information about our subject programs, mutant generation strategies, test case generation strategies, IRs, the standard oracle, and measures.

7.3 Summary

In this thesis, we conducted a mapping study on automated testing techniques that can detect functional software faults in non-testable systems. We believe that this mapping study will be a useful resource for researchers that are attempting to familiarise themselves with/navigate the field. The mapping study highlighted several potential research opportunities, which may serve to steer the direction of future research endeavours.

This thesis also introduced a new testing technique called Interlocutory Testing, which is the first oracle-based approach for alleviating the impact of coincidental correctness on testing. It is our hope that this technique will be useful to practitioners. The thesis also demonstrated that one's method of applying of Interlocutory Testing can be modified to repurpose the technique for other problems. In particular we developed a new technique called Interlocutory Mutation Testing that describes a strategy for applying Interlocutory Testing to alleviate the Equivalent Mutant Problem in the presence of coincidental correctness and/or non-determinism. We hope that practitioners will make use of this technique. We envisage that researchers may also be able to adapt Interlocutory Testing to solve other problems.

This thesis also demonstrated that the theoretical underpinnings of Interlocutory Testing can be integrated into other techniques e.g. Metamorphic Testing (to form Interlocutory Metamorphic Testing) and Spectrum-based Fault Localisation (to form Interlocutory Spectrum-based Fault Localisation), and that the resultant impact of this is an improvement in the effectiveness of these techniques for coincidental correctness. Thus, we conjecture that other's may also be able to integrate the principles of Interlocutory Testing into other techniques to improve the effectiveness of these techniques for coincidental correctness. Again, we hope that practitioners will adopt these techniques.

Chapter 8

References

[1] B. K. Aichernig and E. Jobstl. Efficient Refinement Checking for Model-Based Mutation Testing. In *Proceedings of the 12th International Conference on Quality Software (QSIC)*, pages 21–30, Xi'an, China, 2012. IEEE.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 402–411, Saint Louis, USA, 2005. ACM.

[3] K. Androutsopoulos, D. Clark, H. Dan, R. M. Hierons, and M. Harman. An analysis of the relationship between conditional entropy and failed error propagation in software testing. In *Proceedings of the 36th International Conference on Software Engineering*, pages 573–583, New York, USA, 2014. ACM.

[4] T. Apiwattanapong, R. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold. MATRIX: Maintenance-Oriented Testing Requirements Identifier and Examiner. In *Proceedings of the Testing: Academic and Industrial Conference on Practice And Research Techniques (TAIC PART)*, pages 137–146, Windsor, UK, 2006. IEEE.

[5] W. Araujo, L. C. Briand, and Y. Labiche. On the effectiveness of contracts as test oracles in the detection and diagnosis of functional faults in concurrent object-oriented software. *IEEE Transactions on Software Engineering*, 40:971–992, 2014.

[6] P. Arcaini, A. Gargantini, and E. Riccobene. Combining model-based testing and runtime monitoring for program testing in the presence of nondeterminism. In *Proceedings of the 6th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 178–187, Luxembourg, 2013. IEEE.

[7] M. Asrafi, H. Liu, and F.-C. Kuo. On testing effectiveness of metamorphic relations: A case study. In *Proceedings of the 5th International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*, pages 147–156, Jeju Island, 2011. IEEE.

[8] R. A. Assi, W. Masri, and F. Zaraket. Ucov: User-defined coverage criterion for test case intent verification. Technical report, American University of Beirut, 2014.

[9] R. A. Assi, W. Masri, and F. Zaraket. UCov: a user-defined coverage criterion for test case intent verification. *Software Testing, Verification and Reliability*, 26(6):1–32, 2016.

[10] C. Atkinson, O. Hummel, and W. Janjic. Search-enhanced testing: NIER track. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 880–883, Hawaii, USA, 2011. IEEE.

[11] A. Bandyopadhyay and S. Ghosh. Tester Feedback Driven Fault Localization. In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, pages 41–50, Canada, 2012. IEEE.

[12] L. Baresi and M. Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, 2001.

[13] E. Barnett-Page and J. Thomas. Methods for the synthesis of qualitative research: a critical review. *BMC Medical Research Methodology*, 9(1):1–26, 2009.

[14] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[15] A. C. Barus, T. Y. Chen, D. Grant, F.-C. Kuo, and M. F. Lau. Testing of heuristic methods: A case study of greedy algorithm. In Z. Huzar, R. Koci, B. Meyer, B. Walter, and J. Zendulka, editors, *Software Engineering Techniques*, pages 246–260. Springer Berlin Heidelberg, 2011.

[16] G. Batra and J. Sengupta. An efficient metamorphic testing technique using genetic algorithm. In S. Dua, S. Sahni, and D. P. Goyal, editors, *Information Intelligence, Systems, Technology and Management*, pages 180–188. Springer Berlin Heidelberg, Berlin, Germany, 2011.

[17] M. Bhojasia. Java Program to Implement Knuth Morris Pratt Algorithm. `http://www.sanfoundry.com/java-program-knuth-morris-pratt-algorithm/`, 2013.

[18] S. S. Brilliant, J. C. Knight, and P. E. Ammann. On the performance of software testing using multiple versions. In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing (FTCS-20)*, pages 408–415, Newcastle Upon Tyne, UK, 1990. IEEE.

[19] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, 1982.

[20] Y. Cao, Z. Q. Zhou, and T. Y. Chen. On the correlation between the effectiveness of metamorphic relations and dissimilarities of test case executions. In *Proceedings of the 13th International Conference on Quality Software (QSIC)*, pages 153–162, Najing, China, 2013. IEEE.

[21] R. Carver. Mutation-based testing of concurrent programs. In *Proceedings of the IEEE International Test Conference*, pages 845–853, Baltimore, USA, 1993. IEEE.

[22] F. T. Chan, T. Y. Chen, S. C. Cheung, M. F. Lau, and S. M. Yiu. Application of metamorphic testing in numerical analysis. In *Proceedings of the IASTED International Conference on Software Engineering*, pages 191 – 197, Las Vegas, USA, 1998. ACTA Press.

[23] W. K. Chan, S. C. Cheung, J. C. F. Ho, and T. H. Tse. Reference models and automatic oracles for the testing of mesh simplification software for graphics rendering. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC)*, pages 429–438, Chicago, USA, 2006. IEEE.

[24] W. K. Chan, S. C. Cheung, J. C. F. Ho, and T. H. Tse. PAT: A pattern classification approach to automatic reference oracles for the testing of mesh simplification programs. *Journal of Systems and Software*, 82(3):422–434, 2009.

[25] W. K. Chan, S. C. Cheung, and K. R. P. H. Leung. A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research*, 4(2):61–81, 2007.

[26] W. K. Chan, J. C. F. Ho, and T. H. Tse. Finding failures from passed test cases: improving the pattern classification approach to the testing of mesh simplification programs. *Software Testing, Verification and Reliability*, 20(2):89–120, 2010.

[27] W. K. Chan and T. H. Tse. Oracles are hardly attain'd, and hardly understood: Confessions of software testing researchers. In *Proceedings of the 13th International Conference on Quality Software (QSIC)*, pages 245–252, Najing, China, 2013. IEEE.

[28] H. Y. Chen. The application of an algebraic design method to deal with oracle problem in object-oriented class level testing. In *Proceedings of the International Conference on Systems, Man, and Cybernetics (SMC)*, pages 928–932, Tokyo, Japan, 1999. IEEE.

[29] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: An integrated approach to class-level testing of object-oriented programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(3):250–295, 1998.

[30] H. Y. Chen, T. H. Tse, and Y. T. Deng. ROCS: an object-oriented class-level testing system based on the relevant observable Contexts technique. *Information and Software Technology*, 42(10):677–686, 2000.

[31] J. Chen, F.-C. Kuo, X. Xie, and L. Wang. A cost-driven approach for metamorphic testing. *Journal of Software*, 9(9):2267 − 2275, 2014.

[32] J. Chen, Q. Li, J. Zhao, and X. Li. Test Adequacy Criterion Based on Coincidental Correctness Probability. In *Proceedings of the 2nd Asia-Pacific Symposium on Internetware*, pages 1–4, NY, USA, 2010. ACM.

[33] L. Chen, L. Cai, J. Liu, Z. Liu, S. Wei, and P. Liu. An optimized method for generating cases of metamorphic testing. In *Proceedings of the 6th International Conference on New Trends in Information Science and Service Science and Data Mining (ISSDM)*, pages 439–443, Taipei, Taiwan, 2012. IEEE.

[34] T. Y. Chen. Metamorphic testing: A simple approach to alleviate the oracle problem. In *Proceedings of the 5th IEEE International Symposium on Service Oriented System Engineering (SOSE)*, pages 1–2, Nanjing, China, 2010. IEEE.

[35] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Hong Kong University of Science and Technology, 1998.

[36] T. Y. Chen, J. Feng, and T. H. Tse. Metamorphic testing of programs on partial differential equations: a case study. In *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC)*, pages 327–333, Oxford, UK, 2002. IEEE.

[37] T. Y. Chen, J. W. K. Ho, H. Liu, and X. Xie. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC Bioinformatics*, 10(1):1–12, 2009.

[38] T. Y. Chen, D. H. Huang, T. H. Tse, and Z. Q. Zhou. Case studies on the selection of useful relations in metamorphic testing. In *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC)*, pages 569–583, Australia, 2004. Polytechnic University of Madrid.

[39] T. Y. Chen, F.-C. Kuo, H. Liu, and S. Wang. Conformance testing of network simulators based on metamorphic testing technique. In *Proceedings of the joint 11th IFIP WG 6.1 International Conference FMOODS and 29th IFIP WG 6.1 International Conference FORTE*, pages 243–248, Portugal, 2009. Springer.

[40] T. Y. Chen, F.-C. Kuo, Y. Liu, and A. Tang. Metamorphic testing and testing with special values. In *Proceedings of the 5th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pages 128–134, Beijing, China, 2004. ACIS.

[41] T. Y. Chen, F.-C. Kuo, R. Merkel, and W. K. Tam. Testing an open source suite for open queuing network modelling using metamorphic testing technique. In *Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems*, pages 23–29, Potsdam, Germany, 2009. IEEE.

[42] T. Y. Chen, F.-C. Kuo, W. K. Tam, and R. Merkel. Testing a software-based PID Controller using Metamorphic Testing. In *Proceedings of the 1st International Conference on Pervasive and Embedded Computing and Communication Systems*, pages 387–396, Algarve, Portugal, 2011. SciTePress - Science and Technology Publications.

[43] T. Y. Chen, F.-C. Kuo, D. Towey, and Z. Q. Zhou. Metamorphic testing: Applications and integration with other methods: Tutorial synopsis. In *Proceedings of the 12th International Conference on Quality Software (QSIC)*, pages 285–288, Shaanxi, China, 2012. IEEE.

[44] T. Y. Chen, F.-C. Kuo, T. H. Tse, and Z. Q. Zhou. Metamorphic testing and beyond. In *Proceedings of the 11th Annual International Workshop on Software Technology and Engineering Practice*, pages 94–100, Amsterdam, Netherlands, 2003. IEEE.

[45] T. Y. Chen, P.-L. Poon, and X. Xie. METRIC: METamorphic relation identification based on the category-choice framework. *Journal of Systems and Software*, 116:177 – 190, 2016.

[46] T. Y. Chen, T. H. Tse, and Z. Quan Zhou. Fault-based testing without the need of oracles. *Information and Software Technology*, 45(1):1–9, 2003.

[47] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. Semi-proving: An integrated method for program proving, testing, and debugging. *IEEE Transactions on Software Engineering*, 37(1):109–125, 2011.

[48] D. Clark and R. M. Hierons. Squeeziness: An information theoretic measure for avoiding fault masking. *Information Processing Letters*, 112(8-9):335–340, 2012.

[49] L. M. Connelly. Fisher's Exact Test. *MEDSURG Nursing: Official Journal of the Academy of Medical-Surgical Nurses*, 25(1):58, 2016.

[50] D. S. Cruzes and T. Dyba. Recommended Steps for Thematic Synthesis in Software Engineering. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 275 – 284, Alberta, Canada, 2011. IEEE.

[51] F. Q. B. Da Silva, S. S. J. O. Cruz, T. B. Gouveia, and L. F. Capretz. Using Meta-ethnography to Synthesize Research: A Worked Example of the Relations between Personality and Software Team Processes. In *Proceedings of the ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 153–162, Maryland, USA, 2013. IEEE.

[52] P. Daniel and K. Y. Sim. Debugging in the extreme: Spectrum-based fault localization with limited test cases. *International Journal of Software Engineering and Its Applications*, 7(5):403–412, 2013.

[53] M. D. Davis and E. J. Weyuker. Pseudo-oracles for non-testable programs. In *Proceedings of the ACM 1981 Conference*, pages 254–257, New York, USA, 1981. ACM.

[54] M. E. Delamaro, F. L. S. Nunes, and R. A. P. Oliveira. Using concepts of content-based image retrieval to implement graphical testing oracles. *Software Testing, Verification and Reliability*, 23(3):171–198, 2013.

[55] J. Ding, T. Wu, J. Lu, and X.-H. Hu. Self-checked metamorphic testing of an image processing program. In *Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*, pages 190–197, Singapore, 2010. IEEE.

[56] J. Ding, T. Wu, D. Wu, J. Q. Lu, and X.-H. Hu. Metamorphic testing of a monte carlo modeling program. In *Proceedings of the 6th International Workshop on Automation of Software Test*, pages 1–7, New York, USA, 2011. ACM.

[57] J. Ding and D. Xu. Model-based metamorphic testing: A case study. In *Proceedings of the 24th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 363–368, San Francisco Bay, USA, 2012. Knowledge Systems Institute Graduate School.

[58] G. Dong, T. Guo, and P. Zhang. Security assurance with program path analysis and metamorphic testing. In *Proceedings of the 4th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 193–197, Beijing, China, 2013. IEEE.

[59] G. Dong, C. Nie, B. Xu, and L. Wang. An effective iterative metamorphic testing algorithm based on program path analysis. In *Proceedings of the 7th International Conference on Quality Software (QSIC)*, pages 292–297, China, 2007. IEEE.

[60] G. Dong, B. Xu, L. Chen, C. Nie, and L. Wang. Case studies on testing with compositional metamorphic relations. *Journal of Southeast University*, 24(4):437–443, 2008.

[61] H. Ševčíková, A. Borning, D. Socha, and W.-G. Bleek. Automated testing of stochastic systems: A statistically grounded approach. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 215–224, New York, USA, 2006. ACM.

[62] J. Farjo, R. A. Assi, W. Masri, and F. Zaraket. Does Principal Component Analysis Improve Cluster-Based Analysis? In *Proceedings of the 6th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 400–403, Luxembourg, 2013. IEEE.

[63] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[64] K. Frounchi, L. C. Briand, L. Grady, Y. Labiche, and R. Subramanyan. Automating image segmentation verification and validation by learning test oracles. *Information and Software Technology*, 53(12):1337–1348, 2011.

[65] M. Gligoric, V. Jagannath, and D. Marinov. MuTMuT: Efficient Exploration for Mutation Testing of Multithreaded Code. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, pages 55–64, Washington, DC, USA, 2010. IEEE Computer Society.

[66] T. Goradia. Dynamic Impact Analysis: A Cost-effective Technique to Enforce Error-propagation. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 171–181, NY, USA, 1993. ACM.

[67] R. Gore, P. F. J. Reynolds, and D. Kamensky. Statistical debugging with elastic predicates. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 492–495, Kansas, USA, 2011. IEEE.

[68] A. Gotlieb. Exploiting symmetries to test programs. In *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE)*, pages 365–374, Colorado, USA, 2003. IEEE.

[69] A. Gotlieb and B. Botella. Automated metamorphic testing. In *Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC)*, pages 34–40, Dallas, USA, 2003. IEEE.

[70] R. Guderlei and J. Mayer. Statistical metamorphic testing testing programs with random output by means of statistical hypothesis tests and metamorphic testing. In *Proceedings of the 7th International Conference on Quality Software (QSIC)*, pages 404–409, Oregon, USA, 2007. IEEE.

[71] R. Guderlei and J. Mayer. Towards automatic testing of imaging software by means of random and metamorphic testing. *International Journal of Software Engineering and Knowledge Engineering*, 17(6):757–781, 2007.

[72] R. Guderlei, J. Mayer, C. Schneckenburger, and F. Fleischer. Testing randomized software by means of statistical hypothesis tests. In *Proceedings of the 4th International Workshop on Software Quality Assurance: In Conjunction with the 6th ESEC/FSE Joint Meeting*, pages 46–54, New York, USA, 2007. ACM.

[73] M. Harman and R. M. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[74] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. A comprehensive survey of trends in oracles for software testing. Technical Report TR-09-03, University of Sheffield, 2013.

[75] R. M. Hierons. Avoiding Coincidental Correctness in Boundary Value Analysis. *ACM Transactions on Software Engineering and Methodolology*, 15(3):227–241, 2006.

[76] J. Higgins and S. Green. *Cochrane Handbook for Systematic Reviews of Interventions*. The Cochrane Collaboration, 2011.

[77] D. Hoffman. Heuristic test oracles. *Software Testing & Quality Engineering*, pages 29 – 32, 1999.

[78] S. Huang, M. Y. Ji, Z. W. Hui, and Y. T. Duanmu. Detecting integer bugs without oracle based on metamorphic testing technique. *Applied Mechanics and Materials*, 121-126:1961–1965, 2011.

[79] T. Huckle. Collection of software bugs. `http://www5.in.tum.de/~huckle/bugse.html`, 2011.

[80] Z. Hui, S. Huang, Z. Ren, and Y. Yao. Metamorphic testing integer overflow faults of mission critical program: A case study. *Mathematical Problems in Engineering*, pages 1 – 6, 2013.

[81] O. Hummel, C. Atkinson, D. Brenner, and S. Keklik. Improving testing efficiency through component harvesting. Technical report, University of Mannheim, 2006.

[82] IBM. WALA: T. J. Watson Libraries for Analysis. `http://wala.sourceforge.net/wiki/index.php/Main_Page`, 2015.

[83] W. Janjic, F. Barth, O. Hummel, and C. Atkinson. Discrepancy discovery in search-enhanced testing. In *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*, pages 21–24, New York, USA, 2011. ACM.

[84] M. Jankovic. Genetic Algorithm for Bin Packing Problem. `http://www.codeproject.com/Articles/633133/ga-bin-packing`, 2013.

[85] Java2Novice. Write a program for Bubble Sort in java. - Java Interview Programs. `http://www.java2novice.com/java-interview-programs/bubble-sort/`, 2016.

[86] Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[87] M. Jiang, T. Y. Chen, F.-C. Kuo, and Z. Ding. Testing central processing unit scheduling algorithms using metamorphic testing. In *Proceedings of the 4th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 530–536, Beijing, China, 2013. IEEE.

[88] M. Jiang, T. Y. Chen, F.-C. Kuo, Z. Q. Zhou, and Z. Ding. Testing model transformation programs using metamorphic testing. In *Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 94 – 99, Vancouver, Canada, 2014. Knowledge Systems Institute Graduate School.

[89] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, New York, USA, 2005. ACM.

[90] R. Just and F. Schweiggert. Automating unit and integration testing with partial oracles. *Software Quality Journal*, 19(4):753–769, 2011.

[91] D. Kamensky, R. Gore, and P. F. J. Reynolds. Applying enhanced fault localization technology to monte carlo simulations. In *Proceedings of the 2011 Winter Simulation Conference (WSC)*, pages 2798–2809, Arizona, USA, 2011. IEEE.

[92] U. Kanewala and J. M. Bieman. Techniques for testing scientific programs without an oracle. In *Proceedings of the 5th International Workshop on Software Engineering for Computational Science and Engineering (SE-CSE)*, pages 48–57, California, USA, 2013. IEEE.

[93] U. Kanewala and J. M. Bieman. Using machine learning techniques to detect metamorphic relations for programs without test oracles. In *Proceedings of the 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 1–10, CA, USA, 2013. IEEE.

[94] U. Kanewala and J. M. Bieman. Testing scientific software: A systematic literature review. *Information and Software Technology*, 56(10):1219–1232, 2014.

[95] U. Kanewala, J. M. Bieman, and A. Ben-Hur. Predicting metamorphic relations for testing scientific software: A machine learning approach using graph kernels. *Journal of Software Testing, Verification and Reliability*, pages 1 – 25, 2014.

[96] T. Khosla and S. Garg. Metamorphic testing effectiveness on trignometry. *International Journal of Computer Science and Technology*, 2(3):576–578, 2011.

[97] D. S. Kim-Park, C. Riva, and J. Tuya. A partial test oracle for XML query testing. In *Proceedings of the Testing: Academic and Industrial Conference on Practice and Research Techniques (TAIC PART)*, pages 13–20, Windsor, UK, 2009. IEEE.

[98] D. S. Kim-Park, C. Riva, and J. Tuya. An automated test oracle for XML processing programs. In *Proceedings of the 1st International Workshop on Software Test Output Validation*, pages 5–12, New York, USA, 2010. ACM.

[99] B. Kitchenham. Guidelines for performing Systematic Literature Reviews in Software Engineering. Technical Report EBSE-2007-01, Keele University, 2007.

[100] B. Korel, Q. Zhang, and L. Tao. Assertion-Based Validation of Modified Programs. In *Proceedings of the International Conference on Software Testing Verification and Validation*, pages 426–435, USA, 2009. IEEE.

[101] F.-C. Kuo, S. Liu, and T. Y. Chen. Testing a binary space partitioning algorithm with metamorphic testing. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1482–1489, New York, USA, 2011. ACM.

[102] F.-C. Kuo, Z. Q. Zhou, J. Ma, and G. Zhang. Metamorphic testing of decision support systems: a case study. *IET Software*, 4(4):294–301, 2010.

[103] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process.* Prentice Hall, Upper Saddle River, NJ, 2001.

[104] J. Laski, W. Szermer, and P. Luczycki. Error masking in computer programs. *Software Testing, Verification and Reliability*, 5:81–105, 1995.

[105] Y. Lei, X. Mao, and T. Y. Chen. Backward-slice-based statistical fault localization without test oracles. In *Proceedings of the 13th International Conference on Quality Software (QSIC)*, pages 212–221, Najing, China, 2013. IEEE.

[106] N. Li and J. Offutt. Test Oracle Strategies for Model-Based Testing. *IEEE Transactions on Software Engineering*, 43(4):372 – 395, 2017.

[107] Y. Li and C. Liu. Using Cluster Analysis to Identify Coincidental Correctness in Fault Localization. In *Proceedings of the 4th International Conference on Computational and Information Sciences (ICCIS)*, pages 357–360, Chongqing, China, 2012. IEEE.

[108] Y. Li and C. Liu. Identifying Coincidental Correctness in Fault Localization via Cluster Analysis. *Journal of Software Engineering*, 8(4):328–344, 2014.

[109] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen. How effectively does metamorphic testing alleviate the oracle problem? *IEEE Transactions on Software Engineering*, 40(1):4–22, 2014.

[110] H. Liu, X. Liu, and T. Y. Chen. A new method for constructing metamorphic relations. In *Proceedings of the 12th International Conference on Quality Software (QSIC)*, pages 59–68, Shaanxi, China, 2012. IEEE.

[111] H. Liu, D. Wang, H. Lin, and T. Y. Chen. On the integration of metamorphic testing and model checking. In *Proceedings of the IADIS International Conference Applied Computing*, pages 299–302, Rome, Italy, 2009. IADIS Press.

[112] H. Liu, I. I. Yusuf, H. W. Schmidt, and T. Y. Chen. Metamorphic fault tolerance: An automated and systematic methodology for fault tolerance in the absence of test oracle. In *Proceedings of the 36th International Conference on Software Engineering*, pages 420–423, New York, USA, 2014. ACM.

[113] R. C. Lozano. *Constraint Programming for Random Testing of a Trading System.* PhD thesis, Polytechnic University of Valencia, Stockholm, Sweden, 2010.

[114] X.-l. Lu, Y.-w. Dong, and C. Luo. Testing of component-based software: A metamorphic testing methodology. In *Proceedings of the 7th International Conference on Ubiquitous Intelligence Computing and 7th International Conference on Autonomic Trusted Computing (UIC/ATC)*, pages 272–276, Xian, Shaanxi, 2010. IEEE.

[115] Y. S. Ma, Y. R. Kwon, J. Offutt, and N. Li. Mujava. `http://cs.gmu.edu/~offutt/mujava/`, 2013.

[116] L. I. Manolache and D. G. Kourie. Software testing using model programs. *Software: Practice and Experience*, 31(13):1211–1236, 2001.

[117] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing system virtual machines. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 171–182, New York, USA, 2010. ACM.

[118] W. Masri and R. A. Assi. Cleansing Test Suites from Coincidental Correctness to Enhance Fault-Localization. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST)*, pages 165–174, Paris, France, 2010. IEEE.

[119] W. Masri and R. A. Assi. Prevalence of coincidental correctness and mitigation of its impact on fault localization. *ACM Transactions on Software Engineering and Methodology*, 23(1):1–28, 2014.

[120] W. Masri, R. A. Assi, M. E. Ghali, and N. A. Fatairi. An Empirical Study of the Factors That Reduce the Effectiveness of Coverage-based Fault Localization. In *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 1–5, NY, USA, 2009. ACM.

[121] W. Masri, R. A. Assi, F. Zaraket, and N. Fatairi. Enhancing Fault Localization via Multivariate Visualization. In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, pages 737–741, Canada, 2012. IEEE.

[122] J. Mayer. On testing image processing applications with statistical methods. In *Proceedings of Software Engineering*, pages 69–78, Bonn, Germany, 2005. Lecture Notes in Informatics.

[123] J. Mayer and R. Guderlei. Test oracles using statistical methods. In *Proceedings of the 1st International Workshop on Software Quality*, pages 179–189. Springer, 2004.

[124] J. Mayer and R. Guderlei. An empirical study on the selection of good metamorphic relations. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC)*, pages 475–484, Chicago, USA, 2006. IEEE.

[125] P. McMinn. Search-based failure discovery using testability transformations to generate pseudo-oracles. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, pages 1689–1696, New York, USA, 2009. ACM.

[126] R. Merkel, D. Wang, H. Lin, and T. Y. Chen. Automatic verification of optimization algorithms:: A case study of a quadratic assignment problem solver. *International Journal of Software Engineering and Knowledge Engineering*, 21(2):289–307, 2011.

[127] Y. Miao, Z. Chen, S. Li, Z. Zhao, and Y. Zhou. Identifying Coincidental Correctness for Fault Localization by Clustering Test Cases. Technical report, State Key Laboratory for Novel Software Technology and Nanjing University, 2012.

[128] Y. Miao, Z. Chen, S. Li, Z. Zhao, and Y. Zhou. A clustering-based strategy to identify coincidental correctness in fault localization. *Proceedings of the International Journal of Software Engineering and Knowledge Engineering*, 23(5):721–741, 2013.

[129] K. S. Mishra, G. E. Kaiser, and S. K. Sheth. Effectiveness of teaching metamorphic testing, part II. Technical Report CUCS-022-13, Columbia University, 2013.

[130] T. R. Monisha and A. Chamundeswari. Automatic verification of test oracles in functional testing. In *Proceedings of the 4th International Conference on Computing, Communications and Networking Technologies (ICCCNT)*, pages 1–4, Tiruchengode, India, 2013. IEEE.

[131] C. Murphy and G. E. Kaiser. Improving the dependability of machine learning applications. Technical Report cucs-049-08, Columbia University, 116th and Broadway, New York, NY 10027, 2008.

[132] C. Murphy and G. E. Kaiser. Metamorphic runtime checking of non-testable programs. Technical Report cucs-042-09, Columbia University, 116th and Broadway, New York, NY 10027, 2009.

[133] C. Murphy and G. E. Kaiser. Automatic detection of defects in applications without test oracles. Technical Report CUCS-027-10, Columbia University, 116th and Broadway, New York, NY 10027, 2010.

[134] C. Murphy, G. E. Kaiser, J. S. Bell, and F.-h. Su. Metamorphic runtime checking of applications without test oracles. Technical Report CUCS-023-13, Columbia University, 116th and Broadway, New York, NY 10027, 2013.

[135] C. Murphy, G. E. Kaiser, and L. Hu. Properties of machine learning applications for use in metamorphic testing. In *Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 867–872, California, USA, 2008. SEKE.

[136] C. Murphy, M. S. Raunak, A. King, S. Chen, C. Imbriano, G. Kaiser, I. Lee, O. Sokolsky, L. Clarke, and L. Osterweil. On effective testing of health care simulation software. In *Proceedings of the 3rd Workshop on Software Engineering in Health Care*, pages 40–47, New York, USA, 2011. ACM.

[137] C. Murphy, K. Shen, and G. Kaiser. Automatic system testing of programs without test oracles. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 189–200, New York, USA, 2009. ACM.

[138] C. Murphy, K. Shen, and G. Kaiser. Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles. In *Proceedings of the International Conference on Software Testing Verification and Validation (ICST)*, pages 436–445, Colorado, USA, 2009. IEEE.

[139] P. A. Nardi and M. E. Delamaro. Test oracles associated with dynamical systems models. Technical Report RT 362, Universidade De Sao Paulo, 2011.

[140] D. D. Nardo, N. Alshahwan, L. C. Briand, E. Fourneret, T. Nakic-Alfirevic, and V. Masquelier. Model based test validation and oracles for data acquisition systems. In *Proceedings of the*

28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 540–550, California USA, 2013. IEEE.

[141] A. Núñez and R. M. Hierons. A methodology for validating cloud models using metamorphic testing. *annals of telecommunications - annales des télécommunications*, 70(3):127–135, 2015.

[142] J. A. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*, 1(1):5–20, 1992.

[143] J. A. Offutt. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, 1994.

[144] J. A. Offutt and S. D. Lee. How Strong is Weak Mutation? In *Proceedings of the Symposium on Testing, Analysis, and Verification*, pages 200–213, New York, USA, 1991. ACM.

[145] R. A. P. Oliveira, M. E. Delamaro, and F. L. S. Nunes. Exploring the OFIm framework to support the test of programs with GUIs. Technical report, University of Sao Paulo, 2011.

[146] R. A. P. Oliveira, U. Kanewala, and P. A. Nardi. Automated test oracles: State of the art, taxonomies, and trends. In A. Memon, editor, *Advances in Computers*, pages 113–199. Elsevier, 2014.

[147] R. A. P. Oliveira, A. M. Memon, V. N. Gil, F. L. Nunes, and M. Delamaro. An extensible framework to implement test oracles for non-testable programs. In *Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 199–204, Vancouver, Canada, 2014. Knowledge Systems Institute.

[148] Oracle. Random (Java Platform SE 7). `https://docs.oracle.com/javase/7/docs/api/java/util/Random.html`, 2014.

[149] L. Padgham, Z. Zhang, J. Thangarajah, and T. Miller. Model-based test oracle generation for automated unit testing of agent systems. *IEEE Transactions on Software Engineering*, 39(9):1230–1244, 2013.

[150] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi. N-version disassembly: Differential testing of x86 disassemblers. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 265–274, New York, USA, 2010. ACM.

[151] J. Pallant. *SPSS Survival Manual: A Step-by-Step Guide to Data Analysis using SPSS version 15*. Open University Press, 2007.

[152] M. Papadakis, Y. Jia, M. Harman, and Y. L. Traon. Trivial compiler equivalence: a large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *Proceedings of the 37th International Conference on Software Engineering*, pages 936–946, USA, 2015. IEEE.

[153] G. Paperin. JAGA - Java API for Genetic Algorithms. `http://www.jaga.org/index.html`, 2004.

[154] M. Pezzè and C. Zhang. Automated test oracles: A survey. In A. Memon, editor, *Advances in Computers*, pages 1–48. Elsevier, 2014.

[155] P.-L. Poon, F.-C. Kuo, H. Liu, and T. Y. Chen. How can non-technical end users effectively test their spreadsheets? *Information Technology and People*, 27(4):440–462, 2014.

[156] J. Popay, H. Roberts, A. Sowden, M. Petticrew, L. Arai, M. Rodgers, and N. Britten. *Guidance on the Conduct of Narrative Synthesis in Systematic Reviews: A Product from the ESRC Methods Programme.* Lancaster University, 2006.

[157] D. Poutakidis, L. Padgham, and M. Winikoff. Debugging multi-agent systems using design artifacts: The case of interaction protocols. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems: Part 2*, pages 960–967, New York, USA, 2002. ACM.

[158] D. Poutakidis, M. Winikoff, L. Padgham, and Z. Zhang. Debugging and testing of multi-agent systems using design artefacts. In *Multi-Agent Programming*, pages 215–258. Springer US, Boston, USA, 2009.

[159] R. V. Prasad. Indus. `http://indus.projects.cs.ksu.edu/`, 2015.

[160] L. L. Pullum and O. Ozmen. Early results from metamorphic testing of epidemiological models. In *Proceedings of the ASE/IEEE International Conference on BioMedical Computing (BioMedCom)*, pages 62–67, Washington, USA, 2012. IEEE.

[161] P. Rao, Z. Zheng, T. Y. Chen, N. Wang, and K. Cai. Impacts of test suite's class imbalance on spectrum-based fault localization techniques. In *Proceedings of the 13th International Conference on Quality Software*, pages 260 – 267, Najing, China, 2013. IEEE.

[162] M. Roper. Artifical immune systems, danger theory, and the oracle problem. In *Proceedings of the Testing: Academic and Industrial Conference on Practice and Research Techniques (TAIC PART)*, pages 125–126, Windsor UK, 2009. IEEE.

[163] RosettaCode. Dijkstra's algorithm. `http://rosettacode.org/wiki/Dijkstra%27s_algorithm#Java`, 2015.

[164] J.-N. Rouvignac. AutoRefactor. `https://marketplace.eclipse.org/content/autorefactor`, 2015.

[165] M. S. Sadi, F.-C. Kuo, J. W. K. Ho, M. A. Charleston, and T. Y. Chen. Verification of phylogenetic inference programs using metamorphic testing. *Journal of Bioinformatics and Computational Biology*, 9(6):729–747, 2011.

[166] M. Satpathy, M. Butler, M. Leuschel, and S. Ramesh. Automatic testing from formal specifications. In *Tests and Proofs*, pages 95–113. Springer, 2007.

[167] Scitools. Understand static code analysis tool. `https://scitools.com/`, 2016.

[168] S. Segura, A. Durán, A. B. Sánchez, D. Le Berre, E. Lonca, and A. Ruiz-Cortés. Automated metamorphic testing on the analysis of software variability. Technical Report SA-2013-TR-03, University of Seville, 2013.

[169] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortes. Metamorphic Testing: A Literature Review. Technical Report ISA-16-TR-02, Applied Software Engineering Research Group, University of Seville, Spain, 2016.

[170] S. Segura, R. M. Hierons, D. Benavides, and A. Ruiz-Cortés. Automated metamorphic testing on the analyses of feature models. *Information and Software Technology*, 53(3):245–258, 2011.

[171] E. Shaccour, F. Zaraket, W. Masri, and M. Noureddine. Coverage specification for test case intent preservation in regression suites. In *Proceedings of the 6th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 392 – 395, Luxembourg, 2013. IEEE.

[172] M. Shepperd. Combining Evidence and Meta-analysis in Software Engineering. In A. D. Lucia and F. Ferrucci, editors, *Software Engineering*, pages 46–70. Springer Berlin Heidelberg, 2013.

[173] K. Y. Sim, C. S. Low, and F.-C. Kuo. Detecting faults in technical indicator computations for financial market analysis. In *Proceedings of the 2nd International Conference on Information Science and Engineering (ICISE)*, pages 2749–2754, Hangzhou, China, 2010. IEEE.

[174] K. Y. Sim, C. S. Low, and F.-C. Kuo. Eliminating human visual judgment from testing of financial charting software. *Journal of Software*, 9(2):298–312, 2014.

[175] K. Y. Sim, W. K. S. Pao, and C. Lin. Metamorphic testing using geometric interrogation technique and its application. In *Proceedings of the 2nd International Conference of Electrical Engineering/Electronics, Computer, Telecommunications, and Information Technology (ECTI)*, pages 91–95, 2005.

[176] K. Y. Sim, D. M. L. Wong, and T. Y. Hii. Evaluating the effectiveness of metamorphic testing on Edge detection programs. *International Journal of Innovation, Management and Technology*, 4(1):6–10, 2013.

[177] G. Singh and G. Singh. An automated metamorphic testing technique for designing effective metamorphic relations. In M. Parashar, D. Kaushik, O. F. Rana, R. Samtaney, Y. Yang, and A. Zomaya, editors, *Contemporary Computing*, pages 152–163. Springer Berlin Heidelberg, 2012.

[178] A. B. Sánchez and S. Segura. Automated testing on the analysis of variability-intensive artifacts: An exploratory study with SAT solvers. Technical report, University of Seville, 2012.

[179] Q. Tao, W. Wu, C. Zhao, and W. Shen. An automatic testing approach for compiler based on metamorphic testing technique. In *Proceedings of the 17th Asia Pacific Software Engineering Conference (APSEC)*, pages 270–279, Sydney, Australia, 2010. IEEE.

[180] S. Tiwari, K. K. Mishra, A. Kumar, and A. K. Misra. Spectrum-based fault localization in regression testing. In *Proceedings of the 8th International Conference on Information Technology: New Generations (ITNG)*, pages 191–195, Las Vegas, USA, 2011. IEEE.

[181] T. H. Tse. Research directions on model-based metamorphic testing and verification. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC)*, page 1, Edinburgh, UK, 2005. IEEE.

[182] T. H. Tse, T. Y. Chen, and Z. Zhou. Testing of large number multiplication functions in cryptographic systems. In *Proceedings of the 1st Asia-Pacific Conference on Quality Software*, pages 89–98, Hong Kong, 2000. IEEE.

[183] T. H. Tse, F. C. M. Lau, W. K. Chan, P. C. K. Liu, and C. K. F. Luk. Testing object-oriented industrial software without precise oracles or results. *Communications of the ACM*, 50(8):78–85, 2007.

[184] G. Upton and I. Cook. *A Dictionary of Statistics*. Oxford University Press, 2014.

[185] J. Voas. PIE: a dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, 1992.

[186] Vogella. Sequential and binary search implemented in Java - Tutorial. `http://www.vogella.com/tutorials/JavaAlgorithmsSearch/article.html`, 2009.

[187] X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 45–55, Vancouver, Canada, 2009. IEEE.

[188] L. Weishi and X. Mao. Alleviating the Impact of Coincidental Correctness on the Effectiveness of SFL by Clustering Test Cases. In *Proceedings of the Theoretical Aspects of Software Engineering Conference (TASE)*, pages 66–69, USA, 2014. IEEE Computer Society.

[189] E. J. Weyuker. The oracle assumption of program testing. In *Proceedings of the 13th International Conference on System Sciences (ICSS)*, pages 44 − 49, Hawaii, USA, 1980. Western Periodicals.

[190] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[191] W. E. Wong and V. Debroy. A Survey of Software Fault Localization. Technical Report UTDCS-45-09, The University of Texas at Dallas, 2009.

[192] P. Wu. Iterative metamorphic testing. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC)*, pages 19–24, Edinburgh, UK, 2005. IEEE.

[193] P. Wu, X.-c. Shi, J.-j. Tang, H.-m. Lin, and T. Y. Chen. Metamorphic testing and special case testing: A case study. *Journal of Software*, 16(7):1210 − 1220, 2005.

[194] X. Xie, J. W. K. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software*, 84(4):544–558, 2011.

[195] X. Xie, W. E. Wong, T. Y. Chen, and B. Xu. Spectrum-based fault localization without test oracles. Technical Report UTDCS-07-10, University of Texas at Dallas, 2010.

[196] X. Xie, W. E. Wong, T. Y. Chen, and B. Xu. Metamorphic slice: An application in spectrum-based fault localization. *Information and Software Technology*, 55(5):866–879, 2013.

[197] X. Xue, Y. Pang, and A. Namin. Trimming Test Suites with Coincidentally Correct Test Cases for Enhancing Fault Localizations. In *Proceedings of the 38th Annual Computer Software and Applications Conference (COMPSAC)*, pages 239–244, Vasteras, Sweden, 2014. IEEE.

[198] X. Yang, M. Liu, M. Cao, L. Zhao, and L. Wang. Regression Identification of Coincidental Correctness via Weighted Clustering. In *Proceedings of the 39th Annual Computer Software and Applications Conference (COMPSAC)*, pages 115–120, Taiwan, 2015. IEEE.

[199] X. Yao, M. Harman, and Y. Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proceedings of the 36th International Conference on Software Engineering*, pages 919–930, New York, USA, 2014. ACM.

[200] Y. Yao, S. Huang, and M. Ji. Research on metamorphic testing for oracle problem of integer bugs. In D. Jin and S. Lin, editors, *Advances in Computer Science and Information Engineering*, pages 95–100. Springer Berlin Heidelberg, 2012.

[201] Y. Yi, Z. Changyou, H. Song, and R. Zhengping. Research on metamorphic testing: A case study in integer bugs detection. In *Proceedings of the 4th International Conference on Intelligent Systems Design and Engineering Applications*, pages 488–493, China, 2013. IEEE.

[202] S. Yoo. Metamorphic testing of stochastic optimisation. In *Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, pages 192–201, Paris, France, 2010. IEEE.

[203] A. Zeller and D. Schuler. (Un-)Covering Equivalent Mutants. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, pages 45–54, Paris, France, 2010. IEEE.

[204] J. Zhang, X. Hu, and B. Zhang. An evaluation approach for the program of association rules algorithm based on metamorphic relations. *Journal of Electronics (China)*, 28(4-6):623–631, 2011.

[205] Z. Zhang, W. K. Chan, and T. H. Tse. Fault Localization Based Only on Failed Runs. *IEEE Computer*, 45(6):64–71, 2012.

[206] Z. Zhang, W. K. Chan, T. H. Tse, and P. Hu. Experimental study to compare the use of metamorphic testing and assertion checking. *Journal of Software*, 20(10):2637–2654, 2009.

[207] Z. Zhang, W. K. Chan, T. H. Tse, and B. Jiang. Precise Propagation of Fault-Failure Correlations in Program Flow Graphs. In *Proceedings of the 37th Annual Computer Software and Applications Conference*, pages 58–67, Munich, Germany, 2011. IEEE Computer Society.

[208] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang. Capturing Propagation of Infected Program States. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 43–52, NY, USA, 2009. ACM.

[209] W. Zheng, H. Ma, M. R. Lyu, T. Xie, and I. King. Mining test oracles of web search engines. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 408–411, Kansas, USA, 2011. IEEE.

[210] Z. Zheng, Y. Gao, P. Hao, and Z. Zhang. Coincidental correctness: An interference or interface to successful fault localization? In *Proceedings of the International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 114–119, California, USA, 2013. IEEE.

[211] X. Zhou, L. Wang, X. Li, and J. Zhao. An Empirical Study on the Test Adequacy Criterion Based on Coincidental Correctness Probability. Technical report, State Key Laboratory of Novel Software Technology and Nanjing University, 2014.

[212] Z. Q. Zhou, D. H. Huang, T. H. Tse, Z. Yang, H. Huang, and T. Y. Chen. Metamorphic testing and its applications. In *Proceedings of the 8th International Symposium on Future Software Technology (ISFST)*, pages 1 – 6, Xian, China, 2004. The Software Engineers Association.

[213] Z. Q. Zhou, T. H. Tse, F.-C. Kuo, and T. Y. Chen. Automated functional testing of web search engines in the absence of an oracle. Technical Report TR-2007-06, The University of Hong Kong, 2007.

Chapter 9

Appendices

A Interlocutory Relations - Genetic Algorithm Subject Program

IR1: InitialPopulation

- IOR: Population Size Parameter (input) == pop.size() (output)
  - ID: params.getIndividualsFactory().createRandomIndividual(params) must have been invoked pop.size() number of times

IR2: CreateRandomIndividual

- IOR: The total weight of the solution should be the same as the total weight of DataSet
  - ID: ChosenItem must not be an item that has already been considered
  - ID: DataSet (input) is a permutation of bins (output)
  - ID: The total weight of a specific item id in the solution should be the same as the corresponding item in the dataset
  - ID: DataSet.size() and Solution.size() should be the same (i.e. contain the same number of items)

IR3: CreateRandomIndividualOverflow

- IOR: Maximum Bin Size (input) >= the bin with the most weight (output)
  - ID: When the else statement in the while loop that determines whether a new bin should be spawned is executed, the following condition should be false: (MaximumBinSize-CurrentBin.Size())>=(ItemToBeAdded.size())

IR4: CreateRandomIndividualNewBins

- IOR: getInitialNumberOfBins (input) == bins.size() (output) (before removeEmptyBins() is invoked)
  - ID: The else statement in the while loop that determines whether a new bin should be spawned is never executed

IOR: getInitialNumberOfBins (input) < bins.size() (output) (before removeEmptyBins() is invoked)

ID: The else statement in the while loop that determines whether a new bin should be spawned is executed (bins.size() (output)- bins.size() (input)) number of times

IR5: BinsRemoveEmptyBins

IOR: Bins.size() (input) == Bins.size() (output)

ID: For each bin in Bins, bin.size() > 0

IOR: Bins.size() (input) > Bins.size() (output)

ID: After removing the bin, Bins (before input) should still have the same number of items as Bins (after output)

ID: After removing all empty bins... for each bin in Bins, bin.size() > 0

IR6: FitnessController

IOR: Each member of the population has a fitness value

ID: updateIndividualFitness(Individual indiv, GAParameterSet params) is invoked pop.size() number of times

ID: During the for loop, at an arbitrary iteration, the member of the population under consideration hasn't been checked before

IR7: FindBestFitness

IOR: best (input) < result.getBestFitness() (output)

ID: Add the best (input) and best (output) to the population and perform a sort based on fitness... the last element of the list should be best (output)

IOR: best (input) == result.getBestFitness() (output)

ID: Add the best (input) to the population and perform a sort based on fitness... the last element of the list should be best (input)

IR8: TerminateGA

IOR: terminationConditionApplies() == false

ID: The distance between GenerationNumber and params.getMaxGenerationNumber() should be 1 less than the previous iteration

IOR: terminationConditionApplies() == true

ID: The distance between GenerationNumber and params.getMaxGenerationNumber() should be 1 less than the previous iteration

ID: The distance between the current iteration number and max iterations should be 0

IR9: GAController

IOR: oldPop.size() == newPop.size()

ID: The Crossover operator, Mutation operator, updatePopulationFitness method, the method that finds the individual in the population with the best fitness, and Selection operator should be invoked once by the generateNextPopulation method.

ID: The population size should not have contracted after the execution of the Crossover Operator, Mutation Operator, updatePopulationFitness method, or the method that finds the individual in the population with the best fitness. The population size should also not have expanded after the execution of the Mutation operator, updatePopulationFitness method, the method that finds the individual in the population with the best fitness, and Selection operator. The population size returned by the method should be the same size as the population returned by the Selection Operator.

IR10: AverageFitnessGeneration

IOR: The average fitness of oldPop is less than the average fitness of newPop

ID: The average fitness of the new solutions provided by crossover are above the average of oldPop AND/OR

ID: Mutation managed to mutate some individuals AND getOnlyAcceptMutationIfBetter == true OR

ID: Mutation mutated some individuals and these individuals fitness is greater than they were before (on average across all changed individuals) AND getOnlyAcceptMutationIfBetter == false AND/ OR

ID: Selection deleted individuals that were below average

IOR: The average fitness of oldPop is greater than the average fitness of newPop

ID: The fitness of the new solutions provided by crossover are below the average of oldPop AND/OR

ID: Mutation mutated some individuals and these individuals fitness is less than they were before (on average across all changed individuals) AND getOnlyAcceptMutationIfBetter == false AND/OR

ID: Selection deleted individuals that were above average

IOR: The average fitness of oldPop is equal to the average fitness of newPop

ID: Let Input, Crossover, Mutation and Selection represent the population at that point in time e.g. Crossover = population just after crossover. Let DifferenceInputCrossover, DifferenceCrossoverMutation, DifferenceMutationSelection, DifferenceInputMutation, and DifferenceCrossoverSelection represent the difference in fitness between the two stated elements in each one. Assuming that Crossover and Mutation occurred, the sum of DifferenceInputCrossover, DifferenceCrossoverMutation and DifferenceMutationSelection must be 0. Alternatively, assuming that Crossover did not occur, but Mutation did, the sum of DifferenceInputMutation must be 0. Alternatively, assuming that Crossover occurred, but Mutation did not occur, the sum of DifferenceInputCrossover and DifferenceCrossoverSelection must be 0.

IR11: SelectionController

IOR: The population (input) is the same size as the postapocalyptic population (output)

ID: population size should be the same size as the params.getPopulationSize()

ID: population should be a permutation of postapocalyptic population

ID: select(Population population, int GenerationNumber, GAParameterSet params) should be executed params.getPopulationSize() number of times and output number of times

IOR: The population (input) is larger than the postapocalyptic population (output)

ID: Post apocalyptic population should be == params.getPopulationSize()

ID: postapocalyptic population should be a subset of population

ID: select(Population population, int GenerationNumber, GAParameterSet params) should be invoked gaparameterset.getPopulationSize() number of times and output.size() number of times

IR12: TournamentComposition

IOR: Population.size() == Tournament Size

ID: population.size() == tournament parameter size OR population.size() < tournament parameter size AND the size of the tournament is ! = the tournament parameter size AND the size of the tournament == population.size()

ID: The tournament is a permutation of population

ID: The tournament shouldn't contain any duplicates

ID: Let *Chosen* denote the winner of this tournament, and winners of the previous tournaments. Also let *LeftOvers* denote solutions that have not won a tournament. The union of *Chosen* and *LeftOvers* should be a permutation of the input population.

IOR: Population.size() > the size of the tournament

ID: The tournament should be a subset of population

ID: The size of the tournament == the tournament parameter size

ID: The tournament shouldn't contain any duplicates

ID: Let *Chosen* denote the winner of this tournament, and winners of the previous tournaments. Also let *LeftOvers* denote solutions that have not won a tournament. The union of *Chosen* and *LeftOvers* should be a permutation of the input population.

IR13: AverageTounamentPositionWinner

IOR: The winner of a tournament (output) should be a member of the tournament (input)

ID: On average the members that have an above average fitness in the tournament should win more often than the members with the lowest fitness. For example let Tournament be a set of sorted fitness values: Tournament=1, 1.5, 1.6, 1.9, 2, then members 3 and 4 (by array index convention) have a higher chance of winning than members 0 and 1. Therefore on average, positions 3 and 4 should have more winners than 0 and 1 across all tournaments

IR14: AverageDifferenceSelection

IOR: The average fitness of “selection” (i.e. basically the output population) is higher than the original input population

ID: Then solutions that were moved to the output population from the input population had more of a positive impact i.e. solutions that were added to the population that were above average brought the average up more than solutions that were below average

IOR: The average fitness of “selection” (i.e. basically the output population) is lower than the original input population

ID: inverse of the ID above

IOR: The average fitness of “selection” (i.e. basically the output population) is the same as the original input population

ID: Then solutions that were moved to the output population from the input population had no impact i.e. solutions that were added to the output population that increased the average were counterbalanced by solutions that were added that decreased the average.

IR15: NumberOfParentsControllerLevel

IOR: The input size is less than the output size

ID: getParents(population, params) returned either 2 or more parents

ID: These parents are a subset of the input

IOR: The input size is half the size of the output

ID: getParents(population, params) returned a permutation of the input

ID: The input population size % 2 == 0

IOR: The input size is equal to the output size

ID: getParents(population, params) returned 0 parents

IR16: CrossoverConvergence

IOR: (In the while loop) parents that were used in previous iterations can’t be used in the current iteration... (so only parents that haven’t been considered yet can be selected (and all have an equal opportunity))... all parents must have been considered

ID: The difference between parents.size() and 0 is 2 smaller than was the case in the last iteration and is divisible by 2

IR17: NumberOfChildrenControllerLevel

IOR: Children.size() == (original) parents.size()

ID: CrossOver returned 2 children each time

ID: The while loop was executed Parents.size()/2 times

IR18: ChoosingPairsOfParents

IOR: The output should have an even size

ID: if parents.size()%2 ! = 0 (post for loop) then the if statement responsible for removing a random individual should be invoked and the size of parents.size() should decrease by 1 as a result. Also inverse this. for == 0

IOR: The output size is 0

ID: if parents.size() == 0 (post for loop) OR

ID: if parents.size() == 1 (post for loop)

IR19: ChoosingPairsOfParentsComposition

IOR: The output is a subset of the input

ID: The for loop iterated population.size() number of times

IR20: CrossoverRate

IOR: The crossover rate is 0 and getParents(population, params) returned input.size() number of parents

ID: The number of parents in the population % 2 == 0

IOR: The crossover rate is 0 and getParents(population, params) returned input.size()-1 number of parents

ID: The number of parents in the population % 2 ! = 0

IOR: The crossover rate is 1 and the output is the same size as the input

ID: getParents(population, params) should have returned 0 parents

IOR: (1 > the crossover rate > 0)

ID: on average the number of selected parents should be >= ((1-crossover rate)/2)% of the population. This ID is inconclusive if the input population size is 1.

IR21: ChoosingCouples

IOR: Given a set of parents, they should be paired for reproduction

ID: When choosing two parents to reproduce, these two individuals should not be a member of any other pairing

IR22: CrossOverDuplicate

IOR: Child.size() (input) == Child.size() (output) (OF removeDuplicates)

ID: UnassignedItems.size() is 0

ID: There are no duplicates in the input

IOR: Child.size() (input) > Child.size() (output) (OF removeDuplicates)

ID: Let X be reference to the set of all bins that were removed by removeDuplicates. All unassigned items must be members of X.

ID: Every member in X contains at least one duplicated item

ID: All nonduplicates in X should be a member of unassigneditems

ID: Every bin in X contains at least one item that is in Middle.

- ID: The bins that were removed were not members of middle (unless the exact same bin was duplicated on one of the 2 sides)
- ID: There are duplicates in the input
- ID: There are no duplicates in the output
- ID: The output contains every element that's in the input apart from UnassignedItems and duplicates
- ID: Unassigned items contains no duplicates in the input

IR23: CrossOverController

- IOR: Two children that are together permutations of both parents (in terms of items)
  - ID: makeCopyOfParents, chooseCrossoverPoints, swapBinsBetweenCrossoverPoints must be invoked
  - ID: removeDuplicates, deduceLostItems and reinsertUnassignedItems must be invoked twice

IR24: ChildXPermutationParentX

- IOR: Child 1 is a permutation of Parent 1 (Same works for Child 2)
  - ID: The middle sections that were traded between child 1 and 2 were the same; thus the middle sections of child 1 and child 2 did not change.
- IOR: Child 1 is not a permutation of Parent 1 (Same works for Child 2)
  - ID: The middle sections that were traded between child 1 and 2 were different; thus the middle sections of child 1 and child 2 did change.

IR25: CrossoverClone

- IOR: The output should be a replica of the input
  - ID: Making a change to the output object should not lead to a change in the input object

IR26: ChoosingCrossoverPoints

- IOR: The four selected crossover points must be valid crossover points in their respective children i.e. The first 2 crossover points should be >= 0 and < child1.size() and The second 2 crossover points should be >= 0 and < child2.size()
  - ID: There should only be four crossover points
  - ID: Crossover point 1 <= Crossover point 2 and Crossover point 3 <= Crossover point 4
  - ID: The random number generator has a chance of generating a value between 0 and childx.size()

IR27: SwappingMiddleSection

- IOR: The two children should be a permutation of the two parents combined
  - ID: Child1 should contain two subsequences that are in parent1 and one subsequence from parent2, and vice versa for child2. The start and end of these subsequences should be determined by the crossover points.

IR28: PartitionChild

- IOR: The output must consist of three lists, each of which contains a subset of the input
    - ID: Each list should be a subsequence of the input
    - ID: There should be no duplicates i.e. overlap between these sequences
    - ID: The three lists combined size should equal to the input.size()

IR29: RemoveDuplicates

- IOR: The Prefix shrunk
    - ID: The middle section should be the same
    - ID: The prefix shrunk by the number of duplicates there were across the prefix and middle
    - ID: The prefix contains all of the items that it originally did, except for the duplicates
- IOR: The Prefix stayed the same
    - ID: The middle section should be the same
- IOR: The Suffix shrunk
    - ID: The middle section should be the same
    - ID: The suffix shrunk by the number of duplicates there were across the suffix and middle
    - ID: The suffix contains all of the items that it originally did, except for the duplicates
- IOR: The suffix stayed the same
    - ID: The middle section should be the same

IR30: DeduceLostItems

- IOR: (Child + KnownUnassignedItems + DisplacedItems) should be a permutation of either parent
    - ID: (Child + KnownUnassignedItems + DisplacedItems).size() == DataSet.size()
    - ID: Lost items should be == all items that were in the middle that was traded, but were not returned back in the traded middle

IR31: ReinsertUnassignedItems

- IOR: Child.size() (output) == (Child.size() (input) + UnassignedItems.size())
    - ID: replacementOperationController and FirstFitDecreasing were invoked

IR32: ReplacementOperationController

- IOR: The average size of UnassignedItems should not be larger than in the previous iteration (considers multiple iterations)
    - ID: AnyChanges should be true
- IOR: The average size of UnassignedItems is the same as in the previous iteration (considers last iteration)
    - ID: AnyChanges should be false

IR33: ReplacementOperationControllerUnassignedItems

- IOR: Child (output) is the same as Child (input)

    - ID: UnassignedItems (output) is the same as UnassignedItems (input)
    - ID: replacementOperation must have provided false as output on the first iteration
  - IOR: Child (output) is not the same as Child (input)
    - ID: UnassignedItems (output) is not the same as UnassignedItems (input)
    - ID: UnassignedItems (Output) + Child (Output) is a permutation of UnassignedItems (Input) + Child (Input)
    - ID: replacementOperation must have executed for more than 1 iteration

- IR34: ReplacementLoop
  - IOR: The outer most for loop must iterate UnassignedItems.size() number of times
    - ID: Every item must have been considered once and only once

- IR35: ReplacementLoopNestedBinsLoop
  - IOR: Iterated Child.size() number of times
    - ID: canReplace returned false throughout the entire loop and performReplacement did not execute, or returned canReplace returned true on the last iteration and performReplacement executed on the last iteration
  - IOR: Iterated $<$ Child.size() number of times
    - ID: Let "Iterated" be the iteration canReplace returns true on... canReplace returned false throughout, but returned true on "Iterated", and performReplacement was invoked on this iteration too and no more iterations were performed after that.

- IR36: ReplacementOperationIntegrity
  - IOR: UnassignedItems.size() (input) $<$ UnassignedItems.size() (output)
    - ID: canReplace.getResult()[1].size() $>$ 1 during at least one iteration
    - ID: The child (input) should have more remaining capacity than child (output)
    - ID: None of the childs bins should exceed capacity before, and after the replacement operation has executed
    - ID: The number of items in the child should decrease
    - ID: The number of bins in the child should stay the same
    - ID: ReplacedItems should be a superset of the Child (input) items that were used in replacements
    - ID: Child (output) should not contain any items that are in ReplacedItems
    - ID: Child (output) should contain at least one of the replacement items.
  - IOR: UnassignedItems.size() (input) == UnassignedItems.size() (output) AND Child.GetRemainingCapacity() is smaller
    - ID: canReplace.getResult()[1].size() was at most 1 during the entire loop
    - ID: The child (input) should have more remaining capacity than child (output)
    - ID: None of the childs bins should exceed capacity before, and after the replacement operation has executed

- ID: The number of items in the child should stay the same
- ID: The number of bins in the child should stay the same
- ID: ReplacedItems should be a superset of the Child (input) items that were used in replacements
- ID: Child (output) should not contain any items that are in ReplacedItems
- ID: Child (output) should contain at least one of the replacement items.

IOR: UnassignedItems.size() (input) == UnassignedItems.size() (output) AND Child.GetRemainingCapacity() is the same

- ID: canReplace.getResult()[1] was empty throughout
- ID: AnyChanges should be false under this condition
- ID: None of the child's bins should exceed capacity before, and after the replacement operation has executed
- ID: The number of items in the child should stay the same
- ID: The number of bins in the child should stay the same

IR37: CheckIfCanReplace

IOR: Replacement.size() <= ReplaceXNumberOfItems

- ID: For loop i should be > 0 <= ReplaceXNumberOfItems
- ID: For loop should iterate ReplaceXNumberOfItems number of times OR FoundSuitableReplacement is true
- ID: i should never be the same as i was in any other iteration
- ID: The size of all of the returned Permutations should be == i

IR38: isSuitableForReplacement

IOR: Method returns false

- ID: ((UnassignedItemSize > OnePermutationTotalSize) == false) OR ((((TotalWeightOfBin-OnePermutationTotalSize)+UnassignedItemSize) <= Capacity) == false)

IOR: Method returns true

- ID: Opposite conditions to the first IOR

IR39: FFDIntegrity

IOR: The difference between Child.size() (input) and Child.size() (output) should be unassigneditems size

- ID: (Child (input) + UnassignedItems) should be a permutation of Child (output)
- ID: When the current item under consideration in the for loop is about to be placed into a bin, it should be able to fit this bin and should not be able to fit any of the previous bins.
- ID: After each iteration, child should increases by 1 item
- ID: The item under consideration should not be in child at first, but at the end of the iteration should be in child

- ID: The average spare capacity of the bins on the left hand side should be less (than or equal to) the average spare capacity of the bins on the right hand side

IR40: FFDController

- IOR: The number of bins in the output child == the number of bins in the input child
  - ID: Child.getRemainingCapacity() (input) >= UnassignedItems total weight
  - ID: At no point in the for loop was there a situation where the item under consideration didn't fit in a bin
  - ID: ItemAdded should be true throughout the entire loop
- IOR: The number of bins in the output child > the number of bins in the input child
  - ID: During the for loop, a situation was encountered where the item under consideration couldn't fit in any bin
  - ID: ItemAdded should be == to false on some occasions, thereby forcing the if statement to run... it should run (child.size() (output) minus child.size() (input)) number of times

IR41: SelectedMutants

- IOR: WontMutate.size() + MightMutate.size() == population.size()
  - ID: WontMutate should be a subset of the population
  - ID: MightMutate should also be a subset of the population (before MutateAll is invoked)
  - ID: WontMutate and MightMutate are a permutation of the population
  - ID: WontMutate and MightMutate contain distinct individuals

IR42: ThoseWhoShouldntMutateDidnt

- IOR: The output of MutateAll should be the same size as MightMutate.size()
  - ID: WontMutate (before invocation of MutateAll) is equivalent to WontMutate (after the invocation of MutateAll)

IR43: DecidingWhoShouldMutate

- IOR: The mutation rate is 0 and MightMutate.size() == population.size()
  - ID: WontMutate.size() == 0
- IOR: The mutation rate is 1 and MightMutate.size() == 0
  - ID: WontMutate.size() == population.size()
- IOR: The mutation rate is not 0 or 1
  - ID: Pass

IR44: DecidingWhoShouldMutateFineGrained

- IOR: MightMutate.size() > 0
  - ID: MightMutate.size() + WontMutate.size() == population.size()
  - ID: MightMutate + WontMutate is a permutation of population
  - ID: MutationRate > 0

    - ID: MightMutate.size() == the number of times the dice was rolled > MutationProbability
    - ID: When an item is being added to MightMutate, it must be a member of population, and not a member of WontMutate
    - ID: When an item is being added to WontMutate, it must be a member of population, and not a member of MightMutate
  - IOR: MightMutate.size() == 0
    - ID: WontMutate.size() == population.size()
    - ID: Either the mutation rate == 1 OR
    - ID: the number of times the dice rolled > MutationProbability is 0

- IR45: MutateAllController
  - IOR: MightMutate.size() (input) == MightMutate.size() (output)
    - ID: The second for loop must have considered every member of MightMutate

- IR46: MutateAllIntegrity
  - IOR: MightMutate (input) is a permutation of MightMutate (output)
    - ID: Either all accepted mutations produced solutions that were equivalent to the originals e.g. 1,2,3 and 2,1,3 − > 2,1,3 and 1,2,3
    - ID: OR getOnlyAcceptMutationIfBetter == true AND ShouldUseNewIndividual always returned false
  - IOR: MightMutate (input) only has some common solutions that MightMutate (output) has
    - ID: Let NumberOfMutations be the number of successful mutations. Let count be the number of items that exist in MightMutate (input) that don't exist in MightMutate (output). NumberOfMutations should be >= count
  - IOR: MightMutate (input) contains no solutions that are the same as in MightMutate (output)
    - ID: If all of the solutions were successfully mutated, then none of the mutations were equivalent to anything in the input
    - ID: if getOnlyAcceptMutationIfBetter == true, ShouldUseNewIndividual returned true throughout the entire loop

- IR47: MutateIndividual
  - IOR: The input should be the same size as the output
    - ID: The input should be a permutation of the output
    - ID: The difference between Individual.size() before and after the "selecting bins to delete" should be == getMutationDestroy().
    - ID: The bins in SelectedBins should be a subset of the input and these bins should not exist in ThisIndividualsbins (during "selecting bins to delete")
    - ID: The SelectedBins should not contain duplicates i.e. the same bin shouldn't be selected twice for deletion during ("selecting bins to delete")...
    - ID: reinsertUnassignedItems must execute to put them back in

IR48: ShouldUseNewIndividual

- IOR: The method returns true
  - ID: Mutated has either the same or less spare capacity than Original
- IOR: The method returns false
  - ID: Mutated has either the same or more spare capacity than Original

B Real Faults

In this appendix, we describe the 12 real faults that were found in the Genetic Algorithm subject program, record whether the fault was coincidentally correct, and/or caused a crash during the execution of the SUT. A fault is assumed to be non-coincidentally correct, if it causes such a crash.

Fault 1. The JAGA Developers represented floating point numbers with Double. This caused several rounding errors because Doubles are imprecise [125].

- **Coincidentally correct:** True

- **SUT crashed:** False

Fault 2. Given a set of items, *DataSet*, the BinsFactory class is responsible for generating random solutions based on *DataSet*. Fault 2 caused it to include items that were not in *DataSet*.

- **Coincidentally correct:** False

- **SUT crashed:** False

Fault 3. Let *DataSet* be the set of items to be organised into bins. *DataSet* is intended to be a read-only variable; however some methods were manipulating it, instead of using a copy.

- **Coincidentally correct:** False

- **SUT crashed:** True

Fault 4. The removeEmptyBins method deletes all bins that contain no items, from a solution. This was achieved by iterating over the list of bins in the solutions from left to right and deleting all empty bins that were encountered. The list representation was ArrayList; deletion of elements in an ArrayList causes elements in the list to shuffle to the left and thus offsets the pointer. This meant that, when two consecutive bins were empty the code would delete the first bin, causing the next empty bin to shuffle in its place, and the pointer would be subsequently incremented to the next element; thereby failing to remove the second empty bin.

- **Coincidentally correct:** False

- **SUT crashed:** False

Fault 5. Crossover expands the population, whilst selection contracts it. *PopParam* specifies the size that the population should be at the end of a generation. Thus, there are two strategies; either crossover first expands the population and selection subsequently removes the excess solutions, or selection removes members of the population first and crossover generates solutions to account for the deficit. We opted for the former approach, whilst the JAGA Developers leveraged the latter [153]. The combination of these two approaches meant neither operator would execute because selection wouldn't contract the population unless there was an expansion and crossover wouldn't expand the population unless there was a contraction.

- **Coincidentally correct:** True

- **SUT crashed:** False

Fault 6. Fault 5 was repaired by adopting the former strategy. However, the JAGA toolbox's ordering of these operators had been retained i.e. selection was executed first. This meant that selection didn't execute on the first iteration and the output population of a generation could exceed $PopParam$.

- **Coincidentally correct:** False

- **SUT crashed:** False

Fault 7. Let *Population* be the population that is being subjected to the selection operator and $Survivors$ be a set of individuals that have been selected from *Population* to be passed on to the next generation. During the selection process, once a member of *Population* has been chosen to be a member of $Survivors$, it should not be possible to choose that individual again, because this will create duplicates. However the JAGA Developers code allows for this.

- **Coincidentally correct:** True

- **SUT crashed:** False

Fault 8. In tournament selection, the same individual should not be entered into the same tournament more than once i.e. it shouldn't be able to compete against itself, however, this is possible in the JAGA Developers code.

- **Coincidentally correct:** True

- **SUT crashed:** False

Fault 9. Recall that the JAGA Developers leveraged a strategy, in which the selection operator first removes a set of individuals from the population, and then crossover makes up the deficit. Also recall that we leveraged the opposite strategy i.e. crossover first expands the population, and selection then removes excess individuals. The mechanism that determines the number of individuals to be removed from the population, by the selection operator, was still based the JAGA Developers strategy, instead of the strategy we adopted, and was therefore removing an incorrect number of individuals.

- **Coincidentally correct:** False

- **SUT crashed:** True

Fault 10. Jankovics' design document specifies the use of two point crossover [84]; this involves selecting two parent solutions $P1$ and $P2$, and randomly partitioning each of these solutions into three sublists e.g. $P1_{prefix}$, $P1_{middle}$ and $P1_{suffix}$. The middle partitions of these solutions are finally swapped and the partitions are recombined e.g. $P1 = \{P1_{prefix}, P2_{middle}P1_{suffix}\}$. Jankovic identified that it is possible to encounter scenarios in which an item I exists in either $P1_{prefix}$ or $P1_{suffix}$ and $P2_{middle}$, which means that when the middle partition is swapped, the child that is composed of these three partitions will have duplicate items; thus, he includes the removal of such duplicates in his design [84]. However, since one solution has a duplicate, and in our implementation, bins are moved, not copied, this implies that the other child (from which $P2_{middle}$ was taken, in exchange for $P1_{middle}$) doesn't have I at all. This was not accounted for, and so the integrity of some solutions was compromised.

- **Coincidentally correct:** False

- **SUT crashed:** False

Fault 11. As a part of his design of the MO operator, Jankovic specified a variable called MutationDestroy, which denotes the number of bins in the solution that should be removed, causing the items in these bins to become unassigned (to be reinserted later) [84]. Unfortunately, his design doesn't account for cases where MutationDestroy is greater than the number of bins in the solution. This overlooked detail in the design specification propagated to the SUT.

- **Coincidentally correct:** False

- **SUT crashed:** True

Fault 12. According to Jankovics' design, it's necessary to generate every combination of items in a certain bin (bounded by the GA parameter: $ReplaceXNumberOfItems$) [84]. Each combination must be checked against a certain criterion to identify whether the fitness can be improved by replacing them with an unassigned item. This exhaustive check should be prematurely terminated when a suitable replacement is found; unfortunately this didn't happen.

- **Coincidentally correct:** False

- **SUT crashed:** True

C Interlocutory Relations - Dijkstra's Algorithm

IR1: StartEndNodeRelation

- IOR: Start Node $\equiv$ End Node
  - ID: The outputted path size should be 1 and the node on the path should be $\equiv$ Start Node
  - ID: The total weight of the path should be 0
- IOR: Start Node $\neq$ End Node
  - ID: The outputted path should start with start node and end with end node
  - ID: The path must be valid
  - ID: Let $Edges$ be the set of all edges in the path. The sum of each $Edge \in Edges$ in graph, should be the same as the total weight of the path.
  - ID: The total number of nodes in the path should not exceed the total number of nodes in the graph
  - ID: The path should be a subset of the graph
  - ID: There should be no cycles in the path

IR2: PathGraphRelation

- IOR: The path is a permutation of the graph
  - ID: The size of the path is $\equiv$ to the size of the graph
- IOR: The path is not a permutation of the graph
  - ID: The size of the path $<$ the size of the graph
  - ID: The path should be a subset of the graph

IR3: GraphSizeRelation

- IOR: The graph consists of multiple nodes
  - ID: Let $NumNodes$ denote the total number of nodes in the graph. The algorithm executes $NumNodes$ number of times
  - ID: Visited nodes should increase by 1 and Unvisited nodes should decrease by 1 each iteration
  - ID: The gained/removed nodes by these sets should be the selected node on their respective iterations
  - ID: Visited and Unvisited nodes should always be subsets of the graph
  - ID: The combination of both Visited and Unvisited nodes should always be a permutation of the graph
  - ID: The combination of both Visited and Unvisited nodes should not contain any duplicated elements on any iteration
- IOR: The graph consists of one node
  - ID: The algorithm executes once
  - ID: The outputted path size should be 1 and the element should be $\equiv$ Start Node

ID: The total weight of the path should be 0

ID: Visited nodes should contain one item and unvisited nodes should contain 0 items (post execution)

IR4: PartialPathConstructionRelation

IOR: When a partial path is updated, the new path should be cheaper than the old path

ID: Partial paths must always contain the start node at the start

ID: Partial paths must always be valid

ID: Partial paths must never contain any cycles

ID: Each edge in each partial path must have the same weight as the weight of the corresponding edge in the graph

ID: The selected partial path must be the one with the lowest total weight, on each iteration

ID: If a selected partial path offers a cheaper route to it's neighbour, then it's neighbours previous node should be updated to reflect this. The converse also applies.

D Interlocutory Relations - Bubble Sort

IR1: BubbleSort

- IOR: The output is a permutation of the Input such that the ordering is different
  - ID: At the end of each iteration of the outer for-loop, $n + 1 - m$ number of elements on the right hand side should be sorted
  - ID: There must be $Input.size() \times (Input.size() - 1)$ number of comparisons
  - ID: A swap should only take place if the item under consideration is greater than the adjacent element to its right. The converse is also true.
  - ID: Only adjacent elements should be swapped
- IOR: The output is a permutation of the Input such that ordering is the same
  - ID: At the end of each iteration of the outer for-loop, $n + 1 - m$ number of elements on the right hand side should be sorted
  - ID: There must be $Input.size() \times (Input.size() - 1)$ number of comparisons
  - ID: Every comparison of adjacent pairs should have yielded a "No Swap" decision; the element at position i is less than or equal to the element $i + 1$.

E Interlocutory Relations - Binary Search

IR1: BinarySearch

- IOR: $Input.size() > 0$ and $Output \equiv true$
  - ID: Let $ParentList$ denote a list being cut in half. Also let $Child_1$ and $Child_2$ be the two halves respectively. $Child_1.size() + Child_2.size() \equiv ParentList.size()$ should hold for all partitioning operations
  - ID: $Child_1.size() \approx Child_2.size()$, (tolerance of 1 element acceptable - the left hand side must be greater than the right hand side by 1) in all cases
  - ID: $Sum(List)$ is a function that gets the sum of all elements in $List$. $Sum(ParentList) \equiv Sum(Child_1) + Sum(Child_2)$ in all cases
  - ID: The element of interest should be in the list that has been selected for the next iteration
  - ID: On the last iteration, the last element of $Child_1$ should be the element we are looking for. On all other iterations, the last element of $Child_1$ should not be the element we are looking for
- IOR: $Input.size() > 0$ and $Output \equiv false$
  - ID: Let $ParentList$ denote a list being cut in half. Also let $Child_1$ and $Child_2$ be the two halves respectively. $Child_1.size() + Child_2.size() \equiv ParentList.size()$ should hold for all partitioning operations
  - ID: $Child_1.size() \approx Child_2.size()$, (tolerance of 1 element acceptable - the left hand side must be greater than the right hand side by 1) in all cases
  - ID: $Sum(List)$ is a function that gets the sum of all elements in $List$. $Sum(ParentList) \equiv Sum(Child_1) + Sum(Child_2)$ in all cases
  - ID: On all iterations, the last element of $Child_1$ should not be the element we are looking for
  - ID: On the last iteration, $Child_1$ should not have more than one item left
  - ID: The input list does not contain the input element
- IOR: $Input.size() \equiv 0$ and $Output \equiv false$
  - ID: There should be no iterations of the loop

F Interlocutory Relations - Knuth-Morris-Pratt

IR1: FailureArray

IOR: All values in the failure array are -1

ID: There are no more than one occurences of PATTERN[0] in PATTERN

ID: The for loop in the failure array generator should have executed $PATTERN.length-1$ number of times

ID: If $failure[j] \equiv -1$, then $PATTERN.get(failure[j-1]+1) \neq PATTERN.get(j)$

IOR: There is at least one non -1 value in the failure array

ID: There are more than one occurences of PATTERN[0] in PATTERN

ID: The for loop in the failure array generator should have executed $PATTERN.length-1$ number of times

ID: On a given iteration j, let PREFIX be a subsequence of PATTERN, such that PREFIX contains element 0 to $failure[j]$. Also let SUFFIX be a subsequence of PATTERN, such that SUFFIX contains elements $j - failure[j]$ to j. The PREFIX should be equivalent to the SUFFIX. (only applicable when $failure[j] \neq -1$)

ID: If $failure[j] \equiv -1$, then $PATTERN.get(failure[j-1]+1) \neq PATTERN.get(j)$

IR2: KMPController

IOR: The algorithm located the PATTERN in the TEXT

ID: Let WholeString be a subsequence of TEXT, such that the first element of WholeString is 0 and the last element of WholeString is the END, where END is $x + y - 1$, such that x is the index of the beginning of the pattern found in TEXT and y is the length of the pattern. The PATTERN should be equivalent to elements (x to $x + y - 1$) in WholeString

ID: The value of I should either increase by 1 or stay the same every iteration

ID: On the first iteration, I should be equal to 0, and on the last Iteration I should be equal to $WholeString.size() - 1$

IOR: The algorithm did not locate the PATTERN in the TEXT

ID: The value of I should either increase by 1 or stay the same every iteration

ID: On the first iteration, I should be equal to 0, and on the last Iteration I should be equal to $TEXT.length - 1$

ID: The algorithm should not have executed the if portion of the if statement lenp number of times in a row

IR3: KMPPatternPointer

IOR: Let J_s denote the value of J at the start of the iteration and J_e be the value of J at the end of the iteration. $J_e - J_s \equiv 1$ (the IR assumes that the initial value of j is 0 on the first iteration)

ID: Let PATTERNSEQ be a subsequence of PATTERN, such that PATTERNSEQ is formed from elements 0 to J_s (inclusive). i denotes the value of i at the start of the iteration. Similarly, let TEXTSEQ be a subsequence of TEXT, such that TEXTSEQ is composed of elements $i - J_s$ to i (inclusive). PATTERNSEQ should be equivalent to TEXTSEQ

IOR: $J_e - J_s \equiv 0$

ID: $TEXT.charAt(i) \neq PATTERN.charAt(0)$

IOR: $J_e - J_s < 0$

ID: Let PATTERNSEQ-1 be a subsequence of PATTERN, such that PATTERNSEQ-1 is formed from elements 0 to $j_s - 1$ (inclusive). Similarly, let TEXTSEQ-1 be a subsequence of TEXT, such that TEXTSEQ-1 is composed of elements $i - j_s$ to $i - 1$. PATTERNSEQ-1 should be equivalent to TEXTSEQ-1

ID: PATTERNSEQ should not be equivalent to TEXTSEQ

ID: $j_e - 1 \equiv failure[j_s - 1]$