Extending the Boundaries in Regression Testing: Complexity, Latency, and Expertise

Shin Yoo

Submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy at
King's College London
Department of Computer Science

This thesis is dedicated to my grandmother,

김양희 (金良姬) 1923 – 2007,

to whom I owe so much of what I am.

Abstract

Automated test case management techniques have been studied in order to aid regression testing tasks by reducing their cost and improving their efficiency. However, the current state-of-art test case management techniques remain limited in several aspects. First, the existing techniques share a scalability problem, not only in terms of the size of the test suites and SUT (System Under Test), but also in terms of complexity of the constraints expressible in the process of regression testing, because the problem is often formulated without considering additional constraints. Second, there is no guarantee that the reduced effort does not compromise the fault detection capability of the test suite. Finally, the existing techniques do not provide any means for the human testers to contribute their important domain knowledge. This domain knowledge is hard to capture algorithmically.

This thesis aims to reformulate test case management techniques in these regards by presenting new concepts, algorithms, approaches and combinations of techniques. In order to deal with the complexity of real world regression testing, multi-objective formulation of test case management is presented, allowing the human tester to apply test case management techniques while meeting to multiple objectives. The thesis introduces the concept of latency, which is used to measure the redundancy in a test suite systematically so that the tester can make an informed decision on the appropriate size of test suites. It also shows that the latency of a test suite can be improved automatically using search-based test data augmentation techniques, which can be significantly more efficient compared to existing test data generation techniques. Finally, the thesis considers the combination of clustering and pair-wise comparison approaches to efficiently incorporate the domain knowledge of human tester into test case management.

Acknowledgement

The completion of this thesis would not have been possible without the help and support from many people, to whom I shall be eternally grateful.

First and foremost, I should thank my supervisor, Mark Harman, who, in addition to being the best supervisor one could hope for, made the whole experience so exciting and enjoyable throughout. I am also thankful to my second supervisor, Kathleen Steinhoffel, for providing me with the valuable insights that allowed me to consider my work in different contexts. Andrew Jones not only showed belief in me and supported me when I started the long journey, but also provided me with constructive criticism that helped me to refine my work. I also thank my fellow colleagues in CREST (Centre for Research in Evolution, Search and Testing) for all the advice, help, support and discussions we shared between us. Although I am not naming every one of you, you know who you are.

Dave Binkley always met my blunt questions with friendly, insightful and interesting discussions during his stay at King's College London, for which I am greatly indebted. I should also thank Paolo Tonella and Angelo Susi for their academic insights as well as for their hospitality.

Maggi Lowe and Claudia Mazzoncini from the Department of Computer Science have kindly endured with this administratively challenged student over the years; thank you.

My friends John Mitchell, Keith Rapley and Spiros Michalakopoulos; I thank you for always being there when I felt tired and worn out. Without your friendship this would have been a much tougher journey.

Most of all, I could not even have dreamt of starting all this had I not had the support from my family. I thank my parents, brother and sister and everyone in the family for the love, encouragement and support they have given to me. I am thankful to my parents-in-law for their belief in me. Finally, I thank my loving wife, Mira, for standing beside me through the long journey.

Declaration

The work presented in this thesis was undertaken between October 2006 and April 2009 at King's College London. Parts of this thesis have been published or submitted:

- S. Yoo and M. Harman, Regression Testing Minimisation, Selection and Prioritisation : A Survey. *Software Testing, Verification and Reliability*, *under revision.*
- S. Yoo and M. Harman, Pareto-Efficient Multi-Objective Test Case Selection. *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA)*, London, UK, pages 140-150, 2007.
- S. Yoo, M. Harman and S. Ur, Measuring and Improving Latency to Avoid Test Suite Wear Out, *Proceedings of the 2009 International Workshop on Search-Based Software Testing*, Denver, CO, USA, pages 101-110, 2009. Best Paper Award.
- S. Yoo and M. Harman, Test Data Augmentation: Generating New Test Data from Existing Test Data. *Software Testing, Verification and Reliability*, *under revision*
- S. Yoo, M. Harman, P. Tonella and A. Susi, Clustering Test Cases To Achieve Effective & Scalable Prioritisation Incorporating Expert Knowledge. *Proceedings of the 2009 International Symposium on Software Testing and Analysis (ISSTA)*, Chicago, USA, pages 201-211.

The following papers have also been written during the course of this PhD programme, although they do not form a part of this thesis:

- S. Yoo and M. Harman, Using Hybrid Algorithm For Pareto Efficient Multi-Ob jective Test Suite Minimisation. *Journal of Systems and Software*, *under revision.*
- J. Ren, S. Yoo, M. Harman and J. Krinke, Search Based Data Sensitivity Analysis Applied to Requirement Engineering, *Genetic and Evolutionary Computation Conference 2009 (GECCO 2009)*, pages 1681-1688.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The correctness of software systems is more important than ever as virtually every aspect of modern life, including safety-critical areas, now involves or depends on the role of software. Software failure has a significant impact on economy; a recent report claims that software failure costs United States economy an estimated $59.5 billion, which is approximately 0.6% of US Gross Domestic Product [200].

One way of increasing confidence in the correctness of a software system is through software testing. Software testing is an extremely laborious process; some report that it can account for up to 50% of software development cost [14]. The problem of high testing cost becomes amplified as software life-cycle grows shorter with more competitive market that emphasises shorter *time-to-market* and software development paradigms that introduce shorter development iterations such as the *agile* approach.

Regression testing is a type of software testing that is heavily affected by these factors in particular [121, 122, 124]. Regression testing is a software testing that is performed to increase confidence in the knowledge that newly introduced software features do not obstruct the existing features. Essentially, whenever new features are added to an existing software system, not only the new features should be tested, but also the existing features should be tested to ensure that their behaviours were not affected by the modifications.

This is usually done by applying existing test cases, which test the existing features, to the new system. Therefore, regression testing entails the task of managing a pool of test cases that are repeatedly used for the testing of software systems across multiple versions.

However, as the software system evolves, this pool of test cases are bound to grow larger, thereby increasing the cost of regression testing. Eventually, the pool of test cases will grow so large that the cost of a simple *retest-all* approach becomes inhibitive. For example, a large IBM middleware product is known to require 20,000 test cases for its regression, which take 10 days to run [227]. Automated heuristics have been studied to reduce the cost, either by eliminating the redundancy in the test suites (*test suite minimisation*) [18, 77, 107, 150, 176, 225], selecting only the relevant test cases (*test case selection*) [17, 21, 59, 82, 216, 219], or prioritising test cases so that the fault detection rate is maximised (*test case prioritisation*) [24, 43, 53, 102, 120, 191, 226].

Test suite minimisation is a '*do fewer*' approach; it aims to reduce the size of the given test suite by eliminating the test cases that are redundant in achieving the testing goal such as structural coverage. Test case selection is a '*do smarter*' approach; it first identifies the areas of the System Under Test (SUT) that has been changed, and selects all the test cases that will execute the changed areas. Test suite prioritisation is another '*do smarter*' approach that can be used when the information about changes is unavailable; it aims to prioritise test cases in such an order that will maximise early fault detection. Collectively these techniques form what is called *"test case management"*, in this thesis.

While many heuristics have been developed and studied for test case management, industrial up-take has been slow at best, or non-existent in many cases. This thesis aims to extend the existing test case management techniques in various directions in order to fill the gap between the state-of-art test case management techniques and the real world problems that practitioners face.

The thesis of this dissertation is to reformulate existing regression testing techniques

in order to cope with various aspects of real world challenges in regression testing that have not been considered previously. This is achieved by considering the cross-cutting concerns of multi-objectiveness, redundancy and the use of human expertise. These approaches are evaluated using real examples of software and their test data, providing confidence that the suggested approaches can be successfully deployed to be used in the context of real-world software testing.

1.1 An Illustrative Scenario

This section describes, step by step, a scenario that may develop during regression testing, and how this thesis addresses the issues that may arise.

Suppose that you are a tester in charge of a software product. The product is a mature one with several previous versions already released to the market. You have a large pool of test cases that have been used to test the previous versions. A new version of the software, with a series of new features, is soon to be released. As a tester, one of your tasks is to make sure that the newly introduced features do not interfere with the functionality of the previous versions of the product. For this, you rely on regression testing, which entails executing the accumulated test cases. Fortunately, you have an access to a well-maintained database which keeps track of not only the structural coverage achieved by each test case, but also the faults detected by each test case. What is unfortunate though, is that you only have a limited time allocated for regression testing and the pool of existing test cases is too large to execute in its entirety. In order to finish the regression testing within the given time, you will have to make a few decisions:

- What is the most effective subset of test cases that you can execute within the given time?
- Which test case should you execute first?

You are aware of the so called *test case management* techniques, hitherto presented in the literature, and decide to utilise some of the techniques. In order to finish the regression testing in time, you first consider the use of a test suite minimisation technique, which will produce the minimal subset of test cases that will achieve full structural coverage. However, you also want to utilise the information about the previously detected faults. That is, you want to achieve *fault coverage*, in addition to the structural coverage, by executing all test cases that have a history of detecting faults. Unfortunately, none of the previously published test suite minimisation techniques allow you to consider these two objectives - structural coverage and fault coverage - at the same time. This is the *problem of complexity*, which is described in Section 1.2.1.

You settle for achieving full structural coverage and use the test suite minimisation to obtain the minimal subset of test cases with full structural coverage. The minimisation technique works well. In fact, the minimised subset is so small that you are actually left with some time to do further testing. You decide to apply the minimisation technique once again, to obtain another subset of test cases. However, this turns out to be problematic as the minimisation technique was deterministic and, therefore, the second subset was identical to the first subset. You decide to avoid this problem by applying the minimisation technique to the pool of test cases *after taking out the first subset*. This, however, reveals another problem: some parts of the software are only executed by a very small number of test cases that belonged to the first subset, and there are no alternatives in the rest of the test case pool. Since these test cases execute the most complex functionality of the software, they cost a lot of effort to generate. There is not enough time to generate new test cases for this. This is the *problem of redundancy*, which is described in Section 1.2.2.

Finally, you decide to execute the first subset of test cases. You use a test case prioritisation technique to increase the rate of fault detection and give more time for the programmers to debug. However, you discover that the order produced by the test

case prioritisation technique is not satisfactory: you know that certain test cases are very effective at detecting faults from your experience, but the test case prioritisation technique prioritises these test cases at the end of the order. Adding to the complexity, you are just informed by the marketing department that testing of a particular module in the product needs to be tested first because of a very important business deal the company is about to make. However, there is no way to feed these additional important factors - expertise and other human factors - into the test case prioritisation technique. This is the *problem of expertise*, which is described in Section 1.2.3.

1.2 Problems of this Thesis

This section describes three problems in automated test case management: the problem of complexity, the problem of managing redundancy, and the problem of incorporating expert knowledge. These are generic questions that apply to many aspects of regression testing; this thesis does not intend to claim that the solutions it presents can be fully generalised. It rather attempts to provide the initial momentum in each of these challenges by presenting new approaches and related empirical studies, to support claims for feasibility, performance and applicability.

1.2.1 Problem of Complexity

Any automated test case management technique needs to scale up to the sizes of its real world application. Often the computational complexity of these techniques is based on the number of test cases and the size of the SUT. However, one overlooked aspect of the scalability is the complexity of the problem itself. For example, existing test case prioritisation techniques assume that any permutation of a given set of test cases is feasible for execution, or at least share the same total cost of execution. In reality, this may not be true. There may be dependency relations between test cases, such as the

one between a test case that sets up the database, and another test case that reads from the database. Some test cases may share the same set-up process in a way that reduces the total cost of testing when they are executed side by side. Finally, there may not be enough time to execute the entire test suite according to the given permutation. It is necessary that these additional constraints are considered before automated test case management techniques are widely adopted by practitioners.

This thesis considers the use of multi-objective meta-heuristic optimisation technique to deal with these complex, multiple constraints. Meta-heuristic optimisation techniques have been utilised for test case management before, but the formulations of the problem were all single-objective. The multi-objective formulation will allow us to deal with the inherent complexity of regression testing problems more effectively.

1.2.2 Problem of Managing Redundancy

The aim of test case management is to reduce the effort required for regression testing. However, the question of whether the reduced effort will compromise the fault detection capability of the testing process has not been answered in general. Indeed, it might be inherently impossible to generalise an answer to the question, given the complex interaction between SUT and test suites. There have been contradictory observations on whether eliminating the redundancy in a test suite actually deteriorates its fault detection capability.

Software testing can only show the existence of faults, not the lack of faults. Therefore, provided that there is no constraint on the effort required for testing, redundancy can only be good in a sense that any novel test case will provide increased confidence in the correctness of the SUT. From an engineering viewpoint, the question is how we manage the trade-off between the amount of effort required for testing and the added confidence we can obtain from having the redundancy. Unless the tester is allowed to observe and reason about this trade-off, the threat of compromised fault detection ca-

pability may remain as an obstacle to the adoption of any attempt to reduce the effort of regression testing.

This thesis presents the concept of latency, a systematic measurement of redundancy in a given test suite with respect to a specific testing goal. Having a systematic measurement of redundancy will allow us to represent the trade-off between the effort and fault detection capability. In addition to that, this thesis introduces a meta-heuristic approach to test data augmentation, a technique that generates additional novel test cases from the existing ones so that the desired level of redundancy can be achieved automatically. The cost of test data augmentation is significantly lower than that of existing test data generation techniques. In fact, it is low enough to allow this thesis to introduce a new framework for regression testing called *"on-demand"* regression testing.

1.2.3 Problem of Incorporating Expert Knowledge

A seasoned tester is a source of rich domain knowledge that is hard to capture in algorithms. For example, the human tester may have knowledge about which parts of the SUT are more important or more error-prone; or, the human may have knowledge about which test cases have good history of detecting faults. It would be advantageous for test case management techniques to utilise this expertise. More importantly, the human tester may not want to depend completely on an automated test case management and opt for (or even *insist* upon) making the human tester's own input to the process.

In this regard, incorporating expert knowledge from a human tester is a largely missing element that is essential to the wider adoption of automated test case management. Test case management techniques will benefit not only in terms of their effectiveness and efficiency from the human expertise, but also in a sense that the *blend* of automated techniques and human input allows more smooth transition towards the wider adoption of test case management techniques.

Unfortunately, any human interaction is very costly and, therefore, does not scale

well. This thesis presents a combination of clustering algorithm and a human-based pair-wise comparison approach that is used for test case prioritisation. Clustering test cases based on their similarity reduces the size of the problem, allowing us to utilise a human interactive pair-wise comparison approach.

1.3 Contributions of this Thesis

The contributions of this thesis are as follows:

1. The formulation of test case management as a multi-objective optimisation problem, which is empirically evaluated using both an existing multi-objective genetic algorithm and a novel hybrid algorithm;

2. The demonstration that the multi-objective formulation of the test suite minimisation problem produces solutions that cannot be found by the existing greedy approach, which includes solutions that achieve higher test adequacy at smaller costs;

3. The proposal of the concept of *latency* which allows the tester to systematically measure the redundancy in test suites with respect to the specific testing goal that is present;

4. The introduction of search-based test data augmentation, which is used to automatically enhance the latency of a given test suite by generating novel test data from already existing test data;

5. The empirical evaluation of the search-based test data augmentation against an existing test data generation technique based on local search, which produces evidence of its cost-effectiveness;

6. The introduction of a clustering technique to improve the scalability of human interactive test case prioritisation that is based on pair-wise comparison approach.

1.4 Overview of this Thesis

This thesis is organised as follows:

Chapter 2 - Literature Survey surveys the literature in the area of test case management. The chapter begins by describing the basic concepts in regression testing and test case management. The chapter then considers the literature in three main areas of test case management: test suite minimisation, test case selection and test case prioritisation.

Chapter 3 - Multi-Objective Test Case Management presents the multi-objective formulation of test suite minimisation problem. The chapter first describes the existing single-objective paradigm and a theoretical proof of the $ln(n)$ approximation level of the greedy algorithm for the set cover problem, which is the foundation of test suite minimisation. Then the chapter moves on to the introduction of multi-objective paradigm, and illustrates a multi-objective instance of the test suite minimisation problem in which the performance of a greedy approach is sub-optimal. Finally, the chapter presents the empirical evaluation of the multi-objective optimisation approach in 2-objective and 3-objective formulation of the test suite minimisation problem.

Chapter 4 - Test Suite Latency introduces the concept of test suite latency with its theoretical formulation. Intuitively, latency of a test suite is a measure of its potential to satisfy the given testing goal with different subsets of test cases. The chapter instantiates the latency measurement with respect to structural coverage and investigates various Unix utilities and their test suites. Most of the studied test suites contain a much larger number of test cases than required to achieve the testing goal, i.e. maximum structural coverage. However, they do not provide more than one subset that achieves the goal and, therefore, their latency remains low despite the existence of the redundant test cases. The chapter then presents an automated latency enhancement strategy that is

based on meta-heuristic optimisation and the use of the knowledge of existing test cases. The result shows that the automated approach can successfully enhance the latency of a test suite so that it can provide multiple paths to the achievement of testing goals.

Chapter 5 - Towards On-Demand Regression Testing extends Chapter 4 by suggesting a new approach to regression testing called *on-demand* regression testing. On-demand regression testing tries to overcome the problem of low latency test suites by always providing a novel set of test cases that satisfy a given testing goal. The key to this approach is a low cost, automated test data generation that can guarantee the satisfaction of a specific testing goal. The chapter extends the automated latency enhancement strategy, introduced in Chapter 4, to a technique called test data augmentation. The test data augmentation technique utilises knowledge of existing test data to generate additional test data. The result of the empirical study shows that the cost of test data augmentation can lower than existing search-based test data generation by two orders of magnitude. The fault detection capability of the test data generated by the augmentation technique is evaluated using mutation testing. The result shows that the test data augmentation technique not only maintains competitive level of fault detection capability, but also is capable of detecting different types of mutation faults than the test data generation technique.

Chapter 6 - Use of Expert Knowledge in Test Case Management considers how to resolve the scalability issues that arise when human expert knowledge is utilised for test case management. The chapter studies a test case prioritisation technique that is based on human pair-wise comparison approach. The $O(n^2)$ cost of pair-wise comparison approach is mitigated by the use of clustering: instead of comparing individual test cases, the human expert is required merely to compare clusters of test cases, thereby reducing the load on the human. The actual prioritisation of individual test cases is extrapo-

lated from the human comparisons by interleaving the prioritised clusters. The proposed technique is empirically evaluated using a model of a human tester whose accuracy of comparing test cases for their fault detection capability can vary. The result shows that, for some SUTs, the proposed technique can outperform existing prioritisation techniques based on structural coverage. The chapter also presents an automated test that can be used to determine whether the cost of human input can be justified for the prioritisation of a specific combination of SUT and test suite.

Chapter 7 - Conclusions closes this thesis with summaries of its achievements and proposals of future work.

Chapter 2

Literature Survey

2.1 Background

This section introduces the basic concepts and definitions that form a nomenclature of regression testing and minimisation, selection and prioritisation techniques.

2.1.1 Regression Testing

Regression testing is performed between two different versions of software in order to provide confidence that the newly introduced features of the System Under Test (SUT) do not interfere with the existing features. While the exact details of the modifications made to SUT will often be available, note that they may not be easily available in some cases, such as when the new version is written in a different programming language or when the source code is unavailable.

The following notations are used to describe concepts in the context of regression testing. Let P be the current version of the program under test, and P' be the next version of P. Let S be the current set of specifications for P, and S' be the set of specifications for P'. T is the existing test suite. Individual test cases will be denoted by lower case: t. $P(t)$ stands for the execution of P using t as input.

2.1.2 Distinction between Classes of Techniques

It is necessary at this point to establish a clear terminological distinction between the different classes of techniques described in the paper. Test suite minimisation techniques seek to reduce the size of test suite by eliminating redundant test cases from the test suite. Minimisation is sometimes also called as 'test suite reduction', meaning that the elimination is permanent. However, these two concepts are essentially interchangeable because all reduction techniques can be used to produce a temporary subset of the test suite whereas any minimisation techniques can be used to permanently eliminate test cases. More formally, following Rothermel et al. [165], test suite minimisation is defined as follows:

Definition 1. ***Test Suite Minimisation Problem***

Given: *A test suite, T, a set of test-case requirements $\{r_1, \ldots, r_n\}$, that must be satisfied to provide the desired 'adequate' testing of the program, and subsets of T, $T_1, \ldots, T_n$, one associated with each of the r_is such that any one of the test cases t_j belonging to T_i can be used to achieve requirement r_i.*

Problem: *Find a representative set, T', of test cases from T that satisfies all r_is.*

The testing criterion is satisfied when every test-case requirement in $\{r_1, \ldots, r_n\}$ is satisfied. A test-case requirement, r_i, is satisfied by any test case, t_j, that belongs to T_i, a subset of T. Therefore, the representative set of test cases is the hitting set of T_is. Furthermore, in order to maximise the effect of minimisation, T' should be the minimal hitting set of T_is. The minimal hitting-set problem is an NP-complete problem as is the dual problem of the minimal set cover problem [69].

While test case selection techniques also seek to reduce the size of test suite, the majority of selection techniques are *modification-aware*. That is, the selection is not only

temporary (i.e. specific to the current version of the program), but also focused on the identification of the modified parts of the program. Test cases are selected because they are relevant to the changed parts of SUT, which typically involves a white-box static analysis of the program code. Throughout this chapter, the term 'test case selection problem' is restricted to this modification-aware problem. It is also often referred to as the Regression Test-case Selection (RTS) problem. More formally, following Rothermel and Harrold [171], the selection problem is defined as follows (refer to Section 2.3 for more details on how the subset T' is selected):

Definition 2. ***Test Case Selection Problem***

Given: *The program, P, the modified version of P, P', and a test suite, T.*

Problem: *Find a subset of T, T', with which to test P'.*

Finally, test case prioritisation concerns ordering test cases for early maximisation of some desirable properties, such as the rate of fault detection. It seeks to find the optimal permutation of the sequence of test cases. It does not involve selection of test cases, and assumes that all the test cases may be executed in the order of the permutation it produces, but that testing may be terminated at some arbitrary point during the testing process. More formally, the prioritisation problem is defined as follows:

Definition 3. ***Test Case Prioritisation Problem***

Given: *a test suite, T, the set of permutations of T, PT, and a function from PT to real numbers, $f : PT \rightarrow \mathbb{R}$.*

Problem: *to find $T' \in PT$ such that $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$.*

2.1.3 Classification of Test Cases

Leung and White present the first systematic approach to regression testing by classifying types of regression testing and test cases [121]. Regression testing can be categorised into *progressive* regression testing and *corrective* regression testing. Progressive regression testing involves changes of specifications in P', meaning that P' should be tested against S'. On the other hand, corrective regression testing does not involve changes in specifications, but only in design decisions and actual instructions. It means that the existing test cases can be reused without changing their input/output relation.

Leung and White categorise test cases into five classes. The first three classes consist of test cases that already exist in T.

- **Reusable**: reusable test cases only execute the parts of the program that remain unchanged between two versions, i.e. the parts of the program that are common to P and P'. It is unnecessary to execute these test cases in order to test P'; however, they are called *reusable* because they may still be retained and reused for the regression testing of the future versions of P.

- **Retestable**: retestable test cases execute the parts of P that have been changed in P'. Thus retestable test cases should be re-executed in order to test P'.

- **Obsolete**: test cases can be rendered obsolete because 1) their input/output relation is no longer correct due to changes in specifications, 2) they no longer test what they were designed to test due to modifications to the program, or 3) they are 'structural' test cases that no longer contribute to structural coverage of the program.

The remaining two classes consist of test cases that have yet to be generated for the regression testing of P'.

- **New-structural**: new-structural test cases test the modified program constructs, providing structural coverage of the modified parts in P'.

- **New-specification**: new-specification test cases test the modified program specifications, testing the new code generated from the modified parts of the specifications of P'.

Leung and White go on to propose a retest strategy, in which a *test plan* is constructed based on the identification of changes in the program and classification of test cases. Although the definition of a test plan remains informal, it provides a basis for the subsequent literature; it is especially of more importance to regression test case selection techniques, since these techniques essentially concern the problem of identifying *retestable* test cases. Similarly, test suite minimisation techniques concern the identification of obsolete test cases. Test case prioritisation also can be thought of as a more sophisticated approach to the construction of a test plan.

It should be noted that the subsequent literature focusing on the idea of selecting and reusing test cases for regression testing is largely concerned with corrective regression testing only. For progressive regression testing, it is very likely that new test cases are required in order to test the new specifications. So far this aspect of the overall regression testing picture has been a question mainly reserved for test data generation techniques. However, the early literature envisages a 'complete' regression testing strategy that should also utilise test data generation techniques.

2.2 Test Suite Minimisation

Test suite minimisation techniques aim to identify redundant test cases and to remove them from the test suite in order to reduce the size of the test suite. The minimisation problem described by Definition 1 can be considered as the minimal hitting set problem.

Note that the minimal hitting-set formulation of test suite minimisation problem depends on the assumption that each r_i can be satisfied by a single test case. In practice, this may not be true. For example, suppose that the test-case requirement is functional rather than structural and, therefore, requires more than one test case to be satisfied. The minimal hitting-set formulation no longer holds. In order to apply the given formulation of the problem, a higher level of abstraction would be required so that each test-case requirement can be met with a single test scenario composed of relevant test cases.

2.2.1 Heuristics

The NP-completeness of the test suite minimisation problem encourages the application of heuristics; previous work on test case minimisation can be regarded as the development of different heuristics for the minimal hitting set problem [28, 77, 89, 150].

Horgan and London applied linear programming to the test case minimisation problem in their implementation of a data-flow based testing tool, ATAC [88, 89]. Harrold et al. presented a heuristic for the minimal hitting set problem with the worst case execution time of $O(|T| * max(|T_i|))$ [77]. Here $|T|$ represents the size of the original test suite, and $max(|T_i|)$ represents the cardinality of the largest group of test cases among $T_1, \ldots, T_n$.

Chen and Lau applied GE and GRE heuristics and compared the results to that of HGS (Harrold-Gupta-Soffa) heuristic [28]. The GE and GRE heuristics can be thought of as variations of greedy algorithms, which is known to be an effective heuristic for the set cover problem [154]. Chen et al. defined *essential* test cases as the opposite of *redundnt* test cases. If a test-case requirement r_i can be satisfied by one and only one test case, the test case is an essential test case. On the other hand, if a test case satisfies only a subset of test-case requirements satisfied by another test case, it is a redundant test case. Based on these concepts, the GE and GRE heuristics can be summarised as follows:

- **GE heuristic**: first select all essential test cases in the test suite; for the remaining test-case requirements, use the additional greedy algorithm, i.e. select the test case that satisfies the maximum number of unsatisfied test-case requirements.

- **GRE heuristic**: first remove all redundant test cases in the test suite, which may make some test cases essential; then perform the GE heuristic on the reduced test suite.

Their empirical comparison suggested that no single technique is better than the other, which is natural considering the fact that the techniques concerned are heuristics rather than precise algorithms.

Offutt et al. also treated the test suite minimisation problem as the dual of the minimal hitting set problem, i.e., the set cover problem [150]. Their heuristics can also be thought of as variations of the greedy approach to the set cover problem. However, they adopted several different orderings of consideration of test cases instead of the fixed ordering in the greedy approach. Their empirical study applied their techniques to a mutation-score-based test case minimisation, which reduced sizes of test suites by over 30%.

Whereas other minimisation approaches primarily considered code-level structural coverage, Marré and Bertolino formulated test suite minimisation as a problem of finding a spanning set over a graph [131]. They represented the structure of the SUT using a *decision-to-decision* graph (ddgraph). A ddgraph is a more compact form of the normal CFG since it omits any node that has one entering edge and one exiting edge, making it an ideal representation of the SUT for branch coverage. They also mapped the result of data-flow analysis onto the ddgraph for testing requirements such as *def-use* coverage. Once testing requirements are mapped to entities in the ddgraph, the test suite minimisation problem can be reduced to the problem of finding the minimal spanning set.

Test Case	Testing Requirements					
	r_1	r_2	r_3	r_4	r_5	r_6
t_1	x	x	x			
t_2	x			x		
t_3		x			x	
t_4			x			x
t_5					x	

Table 2.1: Example test suite taken from Tallam and Gupta [198]. The early selection made by the greedy approach, t_1, is rendered redundant by subsequent selections, $\{t_2, t_3, t_4\}$.

Tallam and Gupta developed the greedy approach further by introducing the *delayed greedy* approach, which is based on the Formal Concept Analysis of the relation between test cases and testing requirements [198]. A potential weakness of the greedy approach is that the early selection made by the greedy algorithm can eventually be rendered redundant by the test cases subsequently selected. For example, consider the test suite and testing requirements depicted in Table 2.1, taken from Tallam and Gupta [198]. The greedy approach will select t_1 first as it satisfies the maximum number of testing requirements, and then continues to select t_2, t_3 and t_4. However, after the selection of t_2, t_3 and t_4, t_1 is rendered redundant. Tallam et al. tried to overcome this weakness by constructing a concept lattice, a hierarchical clustering based on the relation between test cases and testing requirements. Tallam et al. performed two types of reduction on the concept lattice. First, if a set of requirements covered by t_i is a superset of the set of requirements covered by t_j, then t_j is removed from the test suite. Second, if a set of test cases that cover requirement r_i is a subset of the set of test cases that cover requirement r_j, requirement r_i is removed. The concept lattice is a natural representation that supports the identification of these test cases. Finally, the greedy algorithm is applied to the transformed set of test cases and testing requirements. In the empirical evaluation, the test suites minimised by this 'delayed greedy' approach were either the same size or smaller than those minimised by the classical greedy approach or by the

HGS heuristic.

Jeffrey and Gupta extended the HGS heuristic so that certain test cases are selectively retained [94, 95]. This 'selective redundancy' is obtained by introducing a secondary set of testing requirements. When a test case is marked as redundant with respect to the first set of testing requirements, Jeffrey and Gupta considered whether the test case is also redundant with respect to the second set of testing requirements. If it is not, the test case is still selected, resulting in a certain level of redundancy with respect to the first set of testing requirements. The empirical evaluation used branch coverage as the first set of testing requirements and *all-uses* coverage information obtained by data-flow analysis. The results were compared to two versions of the HGS heuristic, based on branch coverage and *def-use* coverage. The results showed that, while their technique produced larger test suites, the fault detection capability was better preserved compared to single-criterion versions of the HGS heuristic.

Whereas the selective redundancy approach only considers the secondary criterion when a test case is marked as being redundant by the first criterion, Black et al. considered a bi-criteria approach that takes into account both testing criteria [18]. They combined the *def-use* coverage criterion with the past fault detection history of each test case using a weighted-sum approach and used Integer Linear Programming (ILP) optimisation to find subsets. The weighted-sum approach uses weighting factors to combine multiple objectives. For example, given a weighting factor α and two objectives o_1 and o_2, the new and combined objective, o', is defined as follows:

$$o' = \alpha o_1 + (1 - \alpha) o_2$$

Consideration of a secondary objective using the weighted-sum approach has been used in other minimisation approaches [92] and prioritisation approaches [210]. Hsu and Orso also considered the use of an ILP solver with multi-criteria test suite minimisa-

tion [92]. They extended the work of Black et al. by comparing different heuristics for a multi-criteria ILP formulation: the weighted-sum approach, the prioritised optimisation and a hybrid approach. In prioritised optimisation, the human user assigns a priority to each of the given criteria. After optimising for the first criterion, the result is added as a constraint, while optimising for the second criterion, and so on. However, one possible weakness shared by these approaches is that they require additional input from the user of the technique in the forms of weighting coefficients or priority assignment, which might be biased, unavailable or costly to provide.

The multi-objective formulation of test suite minimisation presented in this thesis also treats time constraint as one of the objectives. For more details, refer to Chapter 3.

McMaster and Memon proposed a test suite minimisation technique based on *call-stack coverage*. A test suite is represented by a set of unique maximum depth call stacks; its minimised test suite is a subset of the original test suite whose execution generates the same set of unique maximum depth call stacks. Note that their approach is different from simply using function coverage for test suite minimisation. Consider two test cases, t_1 and t_2, respectively producing call stacks $c_1 =< f_1, f_2, f_3 >$ and $c_2 =< f_1, f_2 >$. With respect to function coverage, t_2 is rendered redundant by t_1. However, McMaster and Memon regard c_2 to be unique from c_1. For example, it may be that t_2 detects a failure that prevents the invocation of function f_3. Once the call-stack coverage information is collected, the HGS heuristic has been applied. McMaster and Memon later applied the same approach to Graphical User Interface (GUI) testing [133]. It was also implemented for object-oriented systems by Smith et al. [191].

While most of test suite minimisation techniques are based on some kind of coverage criteria, there do exist interesting exceptions. Harder et al. approached test suite minimisation using operational abstraction [72]. An operational abstraction is a formal mathematical description of program behaviour. While it is identical to formal specification in form, an operational abstraction expresses dynamically observed behaviour.

Harder et al. use the widely studied Daikon dynamic invariant detector [55] to obtain operational abstractions. Daikon requires executions of test cases for the detection of possible program invariants. Test suite minimisation is proposed as follows: if the removal of a test case does not change the detected program invariant, it is rendered as redundant. They compred the operational abstraction approach to branch coverage based minimisation. While their approach resulted in larger test suites, it also maintained higher fault detection capability. Moreover, Harder et al. also showed that coverage adequately minimised test suites can be often improved by considering operational abstraction as an additional minimisation criterion.

Leitner et al. propose a somewhat different version of the minimisation problem [119]. They start from the assumption that they already have a *failing test case*, which is too complex and too long for the human tester to understand. Note that this is often the case with randomly generated test data; the test case is often simply too complex for the human to establish the cause of failure. The goal of minimisation is to produce a shorter version of the test case; the testing requirement is that the shorter test case should still reproduce the failure. This minimisation problem is interesting because there is no uncertainty about the fault detection capability; it is given as a testing requirement. Leitner et al. applied the widely studied *delta-debugging* technique [229] to reduce the size of the failing test case.

Schroeder and Korel proposed an approach of test suite minimisation for black-box software testing [188]. They noted that the traditional approach of testing black-box software with combinatorial test suites may result in redundancy since certain inputs to the software may not affect the outcome of the output being tested. They first identified, for each output variable, the set of input variables that can affect the outcome. Then, for each output variable, an individual combinatorial test suite is generated with respect to only those input variables that may affect the outcome. The overall test suite is a union of all combinatorial test suites for individual output variables. Note that this has

a strong connection to the concept of Interaction Testing, which is discussed in detail in Section 2.4.2.

Other work has focused on model-based test suite minimisation [107, 207]. Vaysburg et al. introduced a minimisation technique for model based test suites that uses dependence analysis of Extended Finite State Machines (EFSMs) [207]. Each test case for the model is a sequence of transitions. Through dependence analysis of the transition being tested, it is possible to identify the transitions that affect the tested transition. In other words, testing a transition T can be thought of as testing a set of other transitions that affect T. If a test case tests the same set of transitions as some other, then it is redundant. Korel et al. extended this approach by combining the technique with automatic identification of changes in the model [107]. The dependence analysis based minimisation technique was applied to the set of test cases that were identified to execute the modified transitions. Chen et al. extended Korel's model-based approach to incorporate more complex representations of model changes [30].

A risk shared by most test suite minimisation techniques is that a *discarded* test case may detect a fault. In some domains, however, test suite minimisation techniques can enjoy the certainty of guaranteeing that discarding a test case will not reduce the fault detection capability. Anido et al. investigated test suite minimisation for testing Finite State Machines (FSMs) in this context [6]. When only some components of the SUT need testing, the system can be represented as a composition of two FSMs: *component* and *context*. The context is assumed to be fault-free. Therefore, certain transitions of the system that concern only the context can be identified to be redundant. Under the 'testing in context' assumption (i.e. the context is fault-free), it follows that it is possible to guarantee that a discarded test case cannot detect faults.

Kaminski and Ammann investigated the use of a logic criterion to reduce test suites while guaranteeing fault detection in testing predicates over Boolean vairables [100]. From the formal description of fault classes, it is possible to derive a hierarchy of fault

classes [116]. From the hierarchy, it follows that the ability to detect a class of faults may guarantee the detection of another class. Therefore, the size of a test suite can be reduced by executing only those test cases for the class of faults that subsume another class, whenever this is feasible.

2.2.2 Impact on Fault Detection Capability

Although the techniques discussed so far reported reduced test suites, there has been a persistent concern about the effect that the test suite minimisation has on the fault-detection capability of test suites. Several empirical studies were conducted to investigate this effect [165, 176, 220, 221].

Wong, Horgan, London and Mathur studied ten common Unix programs using randomly generated test suites; this empirical study is often referred to as the WHLM study [220]. To reduce the size of the test suites, they used the ATAC testing tool developed by Horgan and London [88, 89]. First, a large pool of test cases was created using a random test data generation method. From this pool, several test suites with different total block coverage were generated. After generating test suites randomly, artificial faults were seeded into the programs. These artificial faults were then categorised into 4 groups. Faults in Quartile-I can be detected by $[0-25)\%$ of the test cases from the original test suite; the percentage for Quartile-II, III, and IV is $[25-50)\%$, $[50-75)\%$, and $[75-100]\%$ respectably. Intuitively, faults in Quartile-I are harder to detect than those in Quartile-IV. The effectiveness of the minimisation itself was calculated as follows:

$$(1 - \frac{\text{number of test cases in the reduced test suite}}{\text{number of test cases in the original test suite}}) * 100\%$$

The impact of test suite minimisation were measured by calculating the reduction in fault detection effectiveness as follows:

$$(1 - \frac{\text{number of faults detected by the reduced test suite}}{\text{number of faults detected by the original test suite}}) * 100\%$$

By categorising the test suites (by different levels of block coverage) and test cases (by difficulty of detection), they arrived at the following observation. First, the reduction in size is greater in test suites with a higher block coverage in most cases. This is natural considering that test suites with higher block coverage will require more test cases in general. The average size reduction for test suites with (50-55)%, (60-65)%,(70-75)%,(80-85)%, and (90-95)% block coverage was 1.19%, 4.46%, 7.78%, 17.44%, 44.23% respectably. Second, the fault detection effectiveness was decreased by test case reduction, but the overall decrease in fault detection effectiveness is not excessive and could be regarded as worthwhile for the reduced cost. The average effectiveness reduction for test suites with (50-55)%, (60-65)%,(70-75)%,(80-85)%, and (90-95)% block coverage was 0%, 0.03%, 0.01%, 0.38%, 1.45% respectably. Third, test suite minimisation did not decrease the fault detection effectiveness for faults in Quartile-IV at all, meaning all faults in Quartile-IV had been detected by the reduced test suite. The average decrease in fault detection effectiveness for Quartile-I, II, and III was 0.39%, 0.66%, and 0.098% respectably. The WHLM study concluded that, if the cost of testing is directly related to the number of test cases, then the use of the reduction technique is recommended.

Wong, Horgan, London and Pasquini followed up on the WHLM study by applying the ATAC tool to test suites of another, bigger C program; this empirical study is often referred to as the WHLP study [221]. The studied program, `space`, was an interpreter for the ADL(Array Description Language) developed by the European Space Agency. In the WHLP study, test cases were generated, not randomly, but from the operational profiles of `space`; that is, each test case in test case pool was generated so that it matches an example of real usage of `space` recorded in an operational profile. From the test case pool, different types of test suites were generated. The first group of test suites were

constructed by randomly choosing a fixed number of test cases from the test case pool. The second group of test suites were constructed by choosing test cases from the test case pool until a predetermined block coverage target was met. The faults in the program were not artificial, but real faults that were retrieved from development logs.

The results of the WHLP study confirmed the findings of the WHLM study. As in the WHLM study, test suites with low initial block coverage (50%, 55%, 60%, and 65%) showed no decrease in fault detection effectiveness after test suite minimisation. For both the fixed size test suites and fixed coverage test suites, the application of the test case reduction technique did not affect the fault detection effectiveness in any significant way; the average effectiveness reduction due to test suite minimisation was less than 7.28%.

While both the WHLM and WHLP studies showed that the impact of test suite minimisation on fault detection capability was insignificant, other empirical studies produced radically different findings. Rothermel et al. also studied the impact of test suite minimisation on the fault detection capability [176]. They applied the HGS heuristics to the Siemens suite [93], and later expanded this to include `space` [165]. The results from these empirical studies contradicted the previous findings of the WHLM and WHMP studies.

For the study of the Siemens suite [176], Rothermel et al. constructed test suites from the test case pool provided by the Siemens suite so that the test suites include varying amounts of redundant test cases that do not contribute to the decision coverage of the test suite. The effectiveness and impact of reduction was measured using the same metrics that were used in the WHLM study.

Rothermel et al. reported that the application of the test suite minimisation technique produced significant savings in test suite size. The observed tendency in size reduction suggested a logarithmic relation between the original test suite size and the reduction effectiveness, and the results of logarithmic regression confirmed this. The results are repeated here in Table 2.2.

Program	Regression Equation	r^2
`totinfo`	$y = 13.82 \ln x + 30.31$	0.75
`schedule1`	$y = 15.38 \ln x + 24.20$	0.80
`schedule2`	$y = 15.20 \ln x + 25.19$	0.80
`tcas`	$y = 24.31 \ln x - 3.95$	0.88
`printtok1`	$y = 12.76 \ln x + 31.69$	0.80
`printtok2`	$y = 12.11 \ln x + 34.22$	0.78
`replace`	$y = 16.73 \ln x + 7.07$	0.85

Table 2.2: Regression analysis on the relation between the original test suite size(x) and the test suite size reduction effectiveness(y), taken from Rothermel et al. [176].

However, Rothermel et al. also reported that, due to the size reduction, the fault detection capabilities of test suites were severely compromised. The reduction in fault detection effectiveness was over 50% for more than half of 1,000 test suites considered, with some cases reaching 100%. Rothermel et al. also reported that, unlike the size reduction effectiveness, the fault detection effectiveness did not show any particular correlation with the original test suite size.

This initial empirical study was subsequently extended [165]. For the Siemens suite, the results of the HGS heuristic were compared to random reduction by measuring the fault detection effectiveness of randomly reduced test suites. Random reduction was performed by randomly selecting, from the original test suite, the same number of test cases as in the reduced version of the test suite. The results showed that random reduction produced larger decreases in fault detection effectiveness. To summarise the results for the Siemens suite, the test suite minimisation technique produced savings in test suite size, but at the cost of decreased fault detection effectiveness; however, the reduction heuristic showed better fault detection effectiveness than the random reduction technique.

Rothermel et al. also expanded the previous empirical study by including the larger program, `space` [165]. The reduction in size observed in the test suites of `space` confirmed the findings of the previous empirical study of the Siemens suite; the size reduction

effectiveness formed a logarithmic trend, plotted against the original test suite size, similar to the programs in the Siemens suite. More importantly, the reduction in fault detection effectiveness was less than those of the Siemens suite programs. The average reduction in fault detection effectiveness of test suites reduced by the HGS heuristic was 8.9%, while that of test suites reduced by random reduction was 18.1%.

Although the average reduction in fault detection effectiveness is not far from that reported for the WHLP study in the case of `space`, those of the Siemens suite differed significantly from both the WHLP study and the WHLM study, which reported that the application of the minimisation technique did not have significant impact on fault detection effectiveness. Rothermel et al. [165] pointed out the following differences between these empirical studies as candidates for the cause(s) of the contradictory findings, which is paraphrased as follows:

1. Different subject programs: the programs in the Siemens suite are generally larger than those studied in both the WHLM and the WHLP study. Difference in program size and structure certainly could have impact on the fault detection effectiveness.

2. Different types of test suites: the WHLM study used test suites that were not coverage-adequate and much smaller compared to test suites used by Rothermel et al. The initial test pools used in the WHLM study also did not necessarily contain any minimum number of test cases per covered item. These differences could have contributed to less redundancy in test suites, which led to reduced likelihood that test suite minimisation will exclude fault-revealing test cases.

3. Different types of test cases: the test suites used in the WHLM study contained a few test cases that detected all or most of the faults. When such strong test cases are present, reduced versions of the test suites may well show little loss in fault-detection effectiveness.

4. Different types of faults: the faults studied by Rothermel et al. were all Quartile-I

faults according to the definition of the WHLM study, whereas only 41% of the faults studied in the WHLM study belonged to the Quartile-I group. By having more 'easy-to-detect' faults, the test suites used in the WHLM study could have shown less reduction in fault-detection effectiveness after test suite minimisation.

Considering the two contradicting empirical results, it is natural to conclude that the question of evaluating the effectiveness of the test suite minimisation technique is very hard to answer in general and for all testing scenarios. The answer depends on too many factors such as the structure of the SUT, the quality of test cases and test suites, and the types of faults present. This proliferation of potential contributory factors makes it very difficult to generalise any empirical result.

The empirical studies from WHLM, WHLP and Rothermel et al. all evaluated the effectiveness of test suite minimisation in terms of two metrics: percentage size reduction and percentage fault detection reduction. McMaster and Memon noticed that neither metric considers the actual role each testing requirement plays on fault detection [135]. Given a set of test cases, TC, a set of known faults, KF and a set of testing requirements, CR, *fault correlation* for a testing requirement $i \in CR$ to fault $k \in KF$ is defined as follows:

$$\frac{|\{j \in TC | j \text{ covers } i\} \cap \{j \in TC | j \text{ detects } k\}|}{|\{j \in TC | j \text{ covers } i\}|}$$

The expected probability of finding a given fault k after test suite minimisation is defined as the maximum fault correlation of all testing requirements with k. From this, the probability of detecting all known faults in KF is the product of expected probability of finding all $k \in KF$. Since CR is defined by the choice of minimisation criterion, e.g. branch coverage or call-stack coverage, comparing the probability of detecting all known faults provides a systematic method of comparing different minimisation criteria, without depending on a specific heuristic for minimisation. The empirical evaluation of McMaster

and Memon compared five different minimisation criteria for the minimisation of test suites for GUI-intensive applications: event coverage (i.e. each event is considered as a testing requirement), event interaction coverage (i.e. each pair of events is considered as a testing requirement), function coverage, statement coverage and call-stack coverage proposed in [134]. While call-stack coverage achieved the highest average probability of detecting all known faults, McMaster and Memon also found that different faults correlate more highly with different criteria. This analysis provides valuable insights into the selection of minimisation criterion.

Yu et al. considered the effect of test suite minimisation on fault localisation [228]. They applied various test suite minimisation techniques to a set of programs, and measured the impact of the size reduction on the effectiveness of coverage-based fault localisation techniques. Yu et al. reported that higher reduction in test suite size, typically achieved by statement coverage-based minimisation, tends to have a negative impact on fault localisation, whereas minimisation techniques that maintains higher level of redundancy in test suites have negligible impact.

2.3 Test Case Selection

Test case selection, or the regression test selection problem is essentially similar to the test suite minimisation problem; both problems are about choosing a subset of test cases from the test suite. The key difference between these two approaches in the literature is whether the focus is upon the changes in the SUT. Test suite minimisation is often based on metrics such as coverage measured from a single version of the program under test. By contrast, in regression test selection, test cases are selected because their execution is relevant to the changes between the previous and the current version of the SUT.

To recall Definition 2, T' ideally should contain all the *faults-revealing* test cases in T, i.e., the test cases that will reveal faults in P'. In order to define this formally, Rothermel

and Harrold introduced the concept of a $modification - revealing$ test case [169]. A test case t is modification-revealing for P and P' if and only if $P(t) \neq P'(t)$. Given the following two assumptions, it is possible to identify the fault-revealing test cases for P' by finding the modification-revealing test cases for P and P'.

- **P-Correct-for-T Assumption** : It is assumed that, for each test case $t \in T$, when P was tested with t, P halted and produced the correct output.

- **Obsolete-Test-Identification Assumption** : It is assumed that there exists an effective procedure for determining, for each test case $t \in T$, whether t is obsolete for P'.

From these assumptions, it is clear that every test case in T terminates and produces correct output for P, and is also supposed to produce the same output for P'. Therefore, a modification-revealing test case t must be also fault-revealing. Unfortunately, it is not possible to determine whether a test case t is fault-revealing against P' or not because it is undecidable whether P' will halt with t. Rothermel considers a weaker criterion for the selection, which is to select all *modification-traversing* test cases in T. A test case t is modification-traversing for P and P' if and only if :

1. it executes new or modified code in P', or

2. it formerly executed code that has been deleted in P'

Rothermel also introduced the third assumption, which is the Controlled-Regression-Testing assumption.

- **Controlled-Regression-Testing Assumption** : When P' is tested with t, all factors that might influence the output of P', except for the code in P', are kept constant with respect to their states when P was tested with t.

Given that the Controlled-Regression-Testing assumption holds, a non-obsolete test case t can thereby be modification-revealing only if it is also modification-traversing for P and P'. Now, if the P-Correct-for-T assumption and the Obsolete-Test-Identification assumption hold along with the Controlled-Regression-Testing assumption, then the following relation also holds between the subset of fault-revealing test cases, T_{fr}, the subset of modification-revealing test cases, T_{mr}, the subset of modification-traversing test cases, T_{mt}, and the original test suite, T :

$$T_{fr} = T_{mr} \subseteq T_{mt} \subseteq T$$

Rothermel and Harrold admitted that the Controlled-Regression-Testing assumption may not be always practical, since certain types of regression testing may make it impossible to control the testing environment, e.g. testing of a system ported to different operating system [171]. Other factors like non-determinism in programs and time dependencies are also difficult to control effectively. However, finding the subset of modification-traversing test cases may still be useful approach in practice, because T_{mt} is the closest approximation to T_{mr} that can be achieved without executing all test cases. In other words, by finding T_{mt}, it is possible to exclude those test cases that are guaranteed not to reveal any fault in P'. The widely used term, *safe* regression test selection, is based on this concept [167]. A safe regression test selection technique is not safe from all possible faults; however, it is safe in a sense that, if there exists a modification-traversing test case in the test suite, it will definitely be selected.

Based on Rothermel's formulation of the problem, it can be said that test case selection techniques for regression testing focus on identifying the modification-traversing test cases in the given test suite. The details of the selection procedure differ according to how a specific technique defines, seeks and identifies modifications in the program under test. Various techniques have been proposed using different criteria based on, among

others, data flow analysis [71, 76, 81, 197], CFGs (Control Flow Graphs) [167, 168, 170, 173], PDGs (Program Dependence Graphs), SDGs (System Dependence Graphs) [10], program slices [4], and symbolic execution [224]. The following subsections describe these in more detail, highlighting their strengths and weaknesses.

2.3.1 Integer Programming Approach

One of the earliest approaches to test case selection was presented by Fischer and Fischer et al. who used Integer Programming (IP) to represent the selection problem for testing FORTRAN programs [59, 60]. Lee and He implemented a similar technique [118]. Fischer first defined a program segment as a single-entry, single exit block of code whose statements are executed sequentially. Their selection algorithm relies on two matrices that describe the relation between program segments and test cases, as well as between different program segments.

For a program with m segments and n test cases, the IP formulation is given as the problem of finding the decision vector,$< x_1, \ldots, x_n >$, that minimises the following objective function, Z :

$$Z = c_1x_1 + c_2x_2 + \ldots + c_nx_n$$

subject to :

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \geq b_1$$

$$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \geq b_2$$

$$\ldots$$

$$a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \geq b_m$$

The decision vector, $< x_1, \ldots, x_n >$, represents the subset of selected test cases; x_i is equal to 1 if the ith test case is included; 0 otherwise. The coefficients, $c1, \ldots, c_n$, represent the cost of executing each corresponding test case; Fischer et al. used the constant value of 1 for all coefficients, treating all test cases as being equally expensive. The test case dependency matrix, $a_{11}, \ldots, a_{mn}$ represents the relations between test cases and the program segments. The element a_{ij} is equal to 1 if the ith program segment is executed by the test case j; 0 otherwise.

After deriving the series of inequalities, the set of b_k values are determined by using a reachability matrix that describes the program segments that are reachable from other segments. Using this, if one knows the modified segments, it is possible to get all the segments that are reachable from the modified segments, which need to be tested at least once. The integer programming formulation is completed by assigning 1 to the b values for all the segments that need to be tested. The inequality, $a_{i1}x_1 + \ldots + a_{in}x_n \geq b_i$, thus ensures that at least one included test case covers the program element reachable from a change.

Hartmann and Robson implemented and extended a version of Fischer's algorithm in order to apply the technique to C [83–85]. They treat subroutines as segments, achieving subroutine coverage rather than statement coverage.

One weakness in Fischer's approach is its inability to deal with control-flow changes in P'. The test case dependency matrix, $a_{11}, \ldots, a_{mn}$, depends on the control-flow structure of the program under test. If the control-flow structure changes, the test case dependency matrix can be updated only by executing all the test cases, which negates the point of applying the selection technique.

2.3.2 Data-flow Analysis Approach

Several test case selection techniques have been proposed based on data-flow analysis [71, 76, 81, 197]. Data-flow analysis based selection techniques seek to identify new, modified

or deleted definition-use pairs in P', and select test cases that exercise these pairs.

Harrold and Soffa presented data-flow analysis as the testing criterion for an incremental approach to unit testing during the maintenance phase [81]. Taha, Thebaut, and Liu built upon this idea and presented a test case selection framework based on an incremental data-flow analysis algorithm [197]. Harrold and Soffa developed both intra-procedural and inter-procedural selection techniques [76, 82]. Gupta et al. applied program slicing techniques to identify definition-use pairs that are affected by a code modification [71]. The use of slicing techniques enabled identification of definition-use pairs that need to be tested without performing a complete dataflow analysis, which is often very costly. Wong et al. combined data-flow selection approach with coverage-based minimisation and prioritisation to further reduce the effort [219].

One weakness shared by all data-flow analysis-based test case selection techniques is the fact that they are unable to detect modifications that are unrelated to data-flow change. For example, if P' contains new procedure calls without any parameter, or modified output statements that contain no variable uses, data-flow techniques may not select test cases that execute these.

Fisher II et al. applied the data-flow based regression test selection approach for test re-use in spreadsheet programs [61]. Fisher II et al. proposed an approach called What-You-See-Is-What-You-Test (WYSIWYT) to provide incremental, responsive and visual feedback about the *testedness* of cells in spreadsheets. The WYSIWYT framework collects and updates data-flow information incrementally as the user of the spreadsheet makes modifications to cells, using Cell Relation Graph (CRG). Interestingly, the data-flow analysis approach to re-test spreadsheets is largely free from the difficulties that the approach has used to test procedural programs, because spreadsheet programs are purely based on data-flow and not on control-flow information. This makes spreadsheet programs an ideal candidate for a data-flow analysis approach.

2.3.3 Symbolic Execution Approach

Yau and Kishmoto presented a test case selection technique based on symbolic execution of the SUT [224]. In symbolic execution of a program, the variables' values are treated as symbols, rather than concrete values [2]. Yau and Kishimoto's approach can be thought of as an application of symbolic execution and input partitioning to the test case selection problem. First, the technique statically analyses the code and specifications to determine the input partitions. Next, it produces test cases so that each input partition can be executed at least once. Given information on where the code has been modified, the technique then identifies the edges in the control flow graph that lead to the modified code. While symbolically executing all test cases, the technique determines test cases that traverse edges that do not reach any modification. The technique then selects all test cases that reach new or modified code. For the symbolic test cases that reach modifications, the technique completes the execution of those; the real test cases that match these symbolic test cases should be retested.

While it is theoretically powerful, the most important drawback of the symbolic execution approach is the algorithmic complexity of the symbolic execution. Yau and Kishmoto acknowledge that symbolic execution can be very expensive. Pointer arithmetic can also present challenging problems for symbolic execution based approaches.

2.3.4 Dynamic Slicing Based Approach

Agrawal et al. introduced a family of test case selection techniques based on different program slicing techniques [4]. An *execution slice* of a program with respect to a test case is what is usually referred to as an execution trace; it is the set of statements executed by the given test case. A *dynamic slice* of a program with respect to a test case is the set of statements in the execution slice that have an influence on an output statement. Since an execution slice may contain statements that do not affect the program output, a dynamic slice is a subset of an execution slice. For example, consider the program

```
 S1: read(a,b,c);
 S2: class : scalene;
 S3: if a = b or b = a
 S4:      class := isosceles;
 S5: if a * a = b * b + c * c
 S6:      class := right
 S7: if a = b and b = c
 S8:      class := equilateral
 S9: case class of:
S10:      right        : area = b * c / 2;
S11:      equilateral  : area = a * 2 * sqrt(3) / 4;
S12:      otherwise    : s := (a + b + c) / 2;
S13:                     area := sqrt(s * (s-a) * (s-b) * (s-c));
S14: end;
S15: write(class, area);
```

Figure 2.1: Example triangle classification program taken from Agrawal et al. [4]. Note that it is assumed that the input vector is sorted in descending order. It contains two faults. In `S3`, `b = a` should be `b = c`. In `S11`, `a * 2` should be `a * a`.

Testcase	Input			Output	
	a	b	c	class	area
T_1	2	2	2	equilateral	1.73
T_2	4	4	3	isosceles	5.56
T_3	5	4	3	right	6.00
T_4	6	5	4	scalene	9.92
T_5	3	3	3	equilateral	2.60
T_6	4	3	3	scalene	4.47

Table 2.3: Test cases used with the program shown in Figure 2.1, taken from Agrawal et al. [4]. Note that T_5 detects the fault in `S11`, because the value for area should be 3.90. Similarly, T_6 detects the fault in `S3`, because the value for class should be isosceles.

shown in Figure 2.1. It contains two faults in line `S3` and `S11` respectively. The execution slice of the program with respect to test case T_3 in Table 2.3 is shown in Figure 2.2. The dynamic slice of the program with respect to test case T_1 in Table 2.3 is shown in Figure 2.3.

In order to make the selection more precise, Agrawal et al. proposed two additional slicing criteria: a *relevant slice* and an *approximate relevant slice*. A relevant slice of a program with respect to a test case is the dynamic slice with respect to the same test case together with all the predicate statements in the program that, if evaluated

```
 S1: read(a,b,c);
 S2: class : scalene;
 S3: if a = b or b = a
 S4:      class := isosceles;
 S5: if a * a = b * b + c * c
 S6:      class := right
 S7: if a = b and b = c
 S8:      class := equilateral
 S9: case class of:
S10:      right        : area = b * c / 2;
S11:      equilateral  : area = a * 2 * sqrt(3) / 4;
S12:      otherwise    : s := (a + b + c) / 2;
S13:                     area := sqrt(s * (s-a) * (s-b) * (s-c));
S14: end;
S15: write(class, area);
```

Figure 2.2: Execution slice of program shown in Figure 2.1 with respect to test case T_3 in Table 2.3, taken from Agrawal et al. [4]

```
 S1: read(a,b,c);
 S2: class : scalene;
 S3: if a = b or b = a
 S4:      class := isosceles;
 S5: if a * a = b * b + c * c
 S6:      class := right
 S7: if a = b and b = c
 S8:      class := equilateral
 S9: case class of:
S10:      right        : area = b * c / 2;
S11:      equilateral  : area = a * 2 * sqrt(3) / 4;
S12:      otherwise    : s := (a + b + c) / 2;
S13:                     area := sqrt(s * (s-a) * (s-b) * (s-c));
S14: end;
S15: write(class, area);
```

Figure 2.3: Dynamic slice of program shown in Figure 2.1 with respect to test case T_1 in Table 2.3, taken from Agrawal et al. [4]

```
 S1: read(a,b,c);
 S2: class : scalene;
 S3: if a = b or b = a
 S4:       class := isosceles;
 S5: if a * a = b * b + c * c
 S6:       class := right
 S7: if a = b and b = c
 S8:       class := equilateral
 S9: case class of:
S10:       right        : area = b * c / 2;
S11:       equilateral  : area = a * 2 * sqrt(3) / 4;
S12:       otherwise    : s := (a + b + c) / 2;
S13:                       area := sqrt(s * (s-a) * (s-b) * (s-c));
S14: end;
S15: write(class, area);
```

Figure 2.4: Relevant slice of program shown in Figure 2.1 with respect to test case T_4 in Table 2.3, taken from Agrawal et al. [4]

differently, could have caused the program to produce different output. An approximated relevant slice is a more conservative approach to include predicates that could have caused different output; it is the dynamic slice with all the predicate statements in the execution slice. By including all the predicates in the execution slice, an approximated relevant slice caters for the indirect references via pointers. For example, consider the correction of `S3` in the program shown in Figure 2.1. The dynamic slice of T_4 does not include `S3` because the `class` value of T_4 is not affected by any of the lines between `S3` and `S8`. However, the relevant slice of T_4, shown in Figure 2.4, does include `S3` because it could have affected the output when evaluated differently.

The test suite and the previous version of the program under test are preprocessed using these slicing criteria; each test case is connected to a slice, *sl*, constructed by one of the four slicing criteria. After the program is modified, test cases for which *sl* contains the modified statement should be executed again. For example, assume that the fault in `S11`, detected by T_5, is corrected. The program should be retested with T_5. However, T_3 need not be executed because the execution slice of T_3, shown in Figure 2.2, does not contain `S11`. Similarly, assume that the fault in `S3`, detected by T_6, is corrected. The program should be retested with T_6. The execution slice technique selects all six

test cases, T_1 to T_6, after the correction of the fault in `S3` because the execution slices of all six test cases include `S3`. However, it is clear that T_1 and T_3 are not affected by the correction of `S3`; their `class` values are overwritten after the execution of `S3`. The dynamic slicing technique overcomes this weakness. The dynamic slice of T_1 is shown in Figure 2.3. Since `S3` does not affect the output of T_1, it is not included in the dynamic slice. Therefore, the modification of `S3` does not necessitate the execution of T_1.

Agrawal et al. first build their technique on cases in which modifications are restricted to those that do not alter the control flow graph of the program under test. As long as the control flow graph of the program remains the same, their technique is safe and can be regarded as an improvement over Fischer's integer programming approach. Slicing removes the need to formulate the linear programming problem, reducing the effort required from the tester. Agrawal et al. later relaxed the assumption about static control flow graph in order to cater for modifications in the control flow graph of the SUT. If a statement s is added to P, now the slice sl contains all the statements in P that uses the variables defined in s. Similarly, if a predicate p is added to P, the slice sl contains all the statements in P that are control-dependent on p. This does cater for the changes in the control flow graph to some degree, but it is not complete. For example, if the added statement is a simple output statement that does not define or use any variable, then this statement can still be modification-revealing. However, since the new statement does not contain any variable, its addition will not affect any of the existing slices, resulting in an empty selection.

2.3.5 Graph-Walk Approach

Rothermel and Harrold presented regression test case selection techniques based on graph walking of Control Dependence Graphs (CDGs), Program Dependence Graphs (PDGs), System Dependence Graphs (SDGs) and Control Flow Graphs (CFGs) [167, 170, 172, 173]. The CDG is similar to PDG but lacks data dependency relations. By performing a

depth-first traversal of the CDGs of both P and P', it is possible to identify points in a program through which the execution trace reaches the modifications [167]. If a node in the CDG of P is not lexically equivalent to the corresponding node in the CDG of P', the algorithm selects all the test cases that execute the control-dependence predecessors of the mismatching node. The CDG based selection technique does not cater for inter-procedural regression test case selection; Rothermel and Harrold recommend application of the technique at the individual procedural level.

Rothermel and Harrold later extended the graph walking approach to use PDGs for intra-procedural selection, and SDGs for inter-procedural selection [170]. A weakness of the CDG based technique is that, due to the lack of data dependence, the technique will select test cases that execute modified definitions but not the actual uses of a variable. If the modified definition of a variable is never used, it cannot contribute to any different output, and therefore its inclusion is not necessary for safe regression testing. PDGs contain data dependence for a single procedure; SDGs extend this to a complete program with multiple procedures. By using these graphs, Rothermel and Harrold's algorithm is able to check whether a modified definition of a variable is actually used later.

Rothermel and Harrold later presented the graph walking approach based on CFGs [173]. The CFG-based technique essentially follows the approach introduced for the CDG-based technique, but on CFGs rather than on CDGs. Since CFG is a much simpler representation of the structure of a program, the CFG-based technique may be more efficient. However, the CFG lacks data dependence information, so the CFG based technique may select test cases that are not capable of producing different outputs from the original programs as explained above. The technique has been evaluated against various combinations of subject programs and test suites [174]. Ball improved the precision of the graph walk approach with respect to branch coverage [8].

For example, consider Figure 2.5. The algorithm performs a depth-first traversal of two CFGs at the same time. When it visits `S3` and `S3'`, the child nodes `S4` and `S4'` are

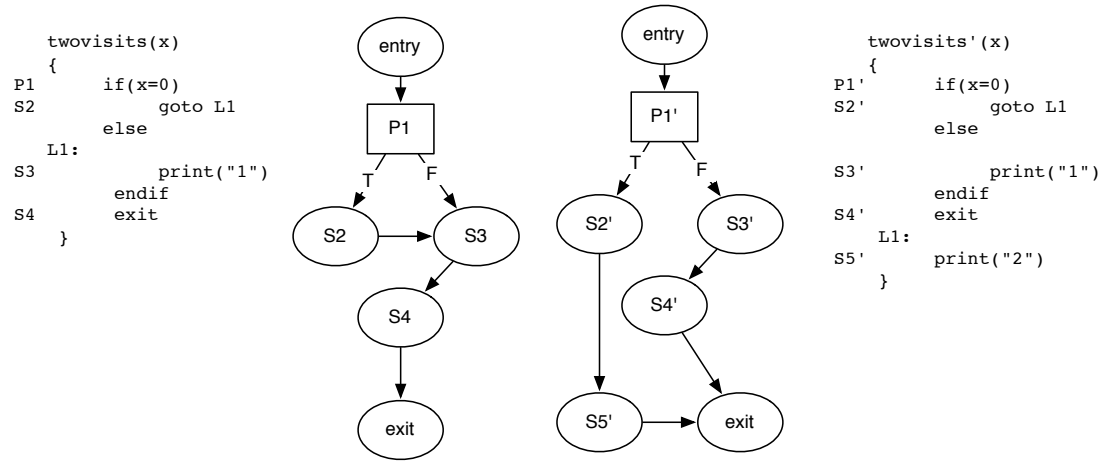

Figure 2.5: Example of CFG-based graph walking algorithm, taken from Rothermel and Harrold [173]. The algorithm detects a modification when comparing `S2` and `S2'`. Since their child nodes are not lexically equivalent to each other, there is a modification. The algorithm then selects all test cases that previously traversed the edge between `S2` and `S3`.

lexically equivalent to each other. However, when it visits `S2` and `S2'`, the child nodes `S3` and `S5'` are lexically different from each other. The algorithm selects all the test cases that previously traversed the edge between `S2` and `S3` in the original program.

Rothermel et al. extended the CFG-based graph walk approach for object-oriented software using the Interprocedural Control Flow Graph (ICFG) [175]. The ICFG connects methods using *call* and *return* edges. Harrold et al. adopted a similar approach for test case selection for Java software, using the Java Interclass Graph as representation (JIG) [79]. Xu and Rountev later extended this technique to consider AspectJ programs by incorporating the interactions between methods and advices at certain join points into the CFG [223]. Zhao et al. also considered a graph representation of AspectJ programs to apply a graph walk approach for RTS [231]. Beydeda and Gruhn extended the graph walk approach by adding black-box data-flow information to the Class Control Flow Graph (CCFG) to test object-oriented software [16].

Orso et al. considered using different types of graph representation of the system to

improve the scalability of graph-walk approach [153]. Their approach first relied on a high-level graph representation of SUT to identify the parts of the system to be further analysed. The technique then used more detailed graph representation to perform more precise selection.

One strength of the graph walk approach is its generic applicability. For example, it has been successfully used in black-box testing of re-usable classes [132]. Martins and Vieira captured the behaviours of a re-usable class by constructing a directed graph called the Behavioural Control Flow Graph (BCFG) from the Activity Diagram (AD) of the class. The BCFG is a directed graph, $G = (V, E, s, x)$, with vertices V, edges E, a unique entry vertex s and an exit vertex x. Each vertex contains a label that specifies the signature of a method; each edge is also labelled according to the corresponding guards in AD. A path in G from s to x represents a possible life history of an object. By mapping changes made to the object to its BCFG and applying the graph walking algorithm, it is possible to select test cases based on the behavioural difference between two versions of the same object. Note that, though, this approach requires traceability between the behavioural model and the actual test cases, because test cases are selected, not based on their structural coverage, but based on their behavioural coverage measured on BCFG. Activity diagrams have also been directly used for RTS by Chen et al. [29].

Orso et al. used a variation of the graph walk approach to consider an RTS technique based on meta-data and specifications obtained from software components [151, 152]. They presented two different techniques based on meta-data: code-based RTS using component meta-data and specification-based RTS using component meta-data. For code-based RTS, it was assumed that each software component was capable of providing structural coverage information, which was fed into the graph walk algorithm. For specification-based RTS, the component specification was represented in UML state-chart diagrams, which were used by the graph walk algorithm.

The graph walk algorithm has also been applied to test web services, despite the

challenges that arise from the distributed nature of web services [127, 181–183, 199]. Several different approaches have been introduced to overcome these challenges. Lin et al. adopted the JIG-based approach after transforming the web services to a single-JVM local application [127]. Ruth et al. collected a coarse-grained CFG from developers of each web service that forms a part of the entire application [181–183]. Finally, Tarhini et al. utilised Timed Labeled Transition System (TLTS), which is a coarse-grained representation of web services that resembles a labelled state machine [199].

2.3.6 Textual Difference Approach

Volkolos and Frankl proposed a selection technique based on the textual difference between the source code of two versions of SUT [208, 209]. They identified modified parts of SUT by applying the `diff` Unix tool to the source code of different versions. The source code was pre-processed into canonical forms to remove the impact of cosmetic differences. Although their technique operates on a different representation of SUT, its behaviour is essentially very similar to that of the CFG based graph-walk approach.

2.3.7 SDG Slicing Approach

Bates and Horwitz proposed test case selection techniques based on program slices from Program Dependency Graphs (PDGs) [10]. Bates and Horwitz approach the regression test selection problem in two stages. First, all the test cases that can be reused for P' need to be identified. Bates and Horwitz introduce the definition of an equivalent execution pattern. If statements s and s' belong to P and P' respectively, s and s' have *equivalent execution patterns* if and only if all of the following hold:

1. For any input file on which both P and P' terminate normally, s and s' are exercised the same number of times.

2. For any input file on which P terminates normally but P' does not, s' is exercised

at most as many times as s is exercised.

3. For any input file on which P' terminates normally but P does not, s is exercised at most as many times as s' is exercised.

Using program slicing, Bates and Horwitz categorize statements into execution classes. Statement s from P and s' from P' belong to the same execution class if and only if any test that exercises s will also exercise s'.

Now, a statement s' in P' is *affected* by the modification if and only if one of the following holds:

1. There is no corresponding statement s in P.

2. The behaviour of s' is not equivalent to the corresponding statement s in P.

Equivalent behaviour is determined by PDG slice isomorphism; if the PDG slice of two statements are isomorphic, then those statements share an equivalent behaviour. For each affected statement in P', reusable test cases are selected based on the information retrieved from the identification stage.

While Bates and Horwitz's technique selects test cases for modified or newly added statements in P', it does not select tests that exercise statements that are deleted from P, and therefore is not safe.

Binkley [17] presented a technique based on System Dependence Graph (SDG) slicing, which extends Bates and Horwitz's intra-procedural to inter-procedural test case selection. Binkley introduced the concept of *common execution patterns*, which corresponds to the equivalent execution patterns of Bates and Horwitz.

2.3.8 Path Analysis

Benedusi et al. applied path analysis for test case selection [15]. They construct *exemplar paths* from P and P' expressed in an algebraic expression. By comparing two sets of

exemplar paths, they classified paths in P' as new, modified, cancelled, or unmodified. Test cases and the paths they execute in P are known; therefore, they selected all the test cases that will traverse modified paths in P'.

One potential weakness of the path analysis approach of Benedusi et al. lies not in path analysis itself, but in the potentially over-specific definition of 'modification' used in the post-analysis selection phase. No test cases are selected for the paths that are classified to be new or cancelled. However, new or cancelled paths denote modifications that represent differences between P and P'; test cases that execute new or cancelled paths in P' may be modification-revealing. As presented, therefore, the path analysis approach is not safe.

2.3.9 Modification-based Technique

Chen et al. introduced a testing framework called TestTube, which utilises a modification-based technique to select test cases [31]. TestTube partitions the SUT into *program entities*, and monitors the execution of test cases to establish connections between test cases and the program entities that they execute. TestTube also partitions P' into program entities, and identifies program entities that are modified from P. All the test cases that execute the modified program entities in P should be re-executed.

TestTube can be thought of as an extended version of the graph walk approach. Both techniques identify modifications by examining the program source code, and select test cases that will execute the modified parts. TestTube extends the CDG-based graph walk technique by introducing program entities that include both functions and entities that are not functions, i.e. variables, data types, and pre-processor macros. Any test case that executes modified functions will be selected. Therefore, TestTube is a safe test case selection technique.

One weakness of TestTube is pointer handling. By including variable and data types

as program entities, TestTube requires that all value creations and manipulations in a program can be inferred from source code analysis. This is only valid for languages without pointer arithmetic and type coercion. As a result, TestTube makes assumptions such as all pointer arithmetic is well-bounded. If these assumptions do not hold then safety cannot be guaranteed.

2.3.10 Firewall Approach

Leung and White introduced and later implemented what they called a *firewall* technique for regression testing of system integration [122, 123, 216, 217]. The main concept is to draw a firewall around the modules of the system that need to be retested. They categorise modules into the following categories:

- No Change : module A has not been modified, $NoCh(A)$.
- Only Code Change : module A has the same specification but its code has been modified, $CodeCh(A)$.
- Spec Change : module A has modified specifications, $SpecCh(A)$.

If a module A calls a module B, there exist 9 possible pairings between the states of A and B. The integration between A and B can be ignored for regression testing if $NoCh(A) \wedge NoCh(B)$, leaving 8 pairings. If both A and B are modified either in code or in specifications, the integration tests between A and B should be executed again as well as the unit tests of A and B; this accounts for 4 of the remaining 8 pairings. The other 4 pairings are cases in which an unchanged module calls a changed module, or vice versa; these pairs form the boundary for integration testing, i.e. the so-called firewall.

By considering modules as the atomic entities, Leung and White maintained a very conservative approach to test case selection. If a module has been modified, any test

case that tests the integration of the modified module should be selected. Therefore, all modification-traversing test cases will be selected. However, their technique may also select other test cases that execute the modified module, but are not modification-traversing in any way. Leung and White also noted that, in practice, the test suite for system integration is often not very reliable. The low reliability means that it is more likely that there may still exist a fault-revealing test case that does not belong to the test suite, and therefore cannot be selected. Note that it is always a risk that a fault-revealing test case exists outside the given test suite in any type of testing, not only in integration testing. What Leung and White pointed out was that such a risk can be higher in system integration testing due to the generally low quality of test suites.

The Firewall approach has been applied to Object-Oriented programs [113, 215, 218] and GUIs [214]. Firewall approach has also been successfully applied to RTS for black-box Commercial Off-the-Shelf (COTS) components. Zheng et al. applied the firewall technique of Leung and White based on the information extracted from the deployed binary code [232–235]. Skoglund and Runeson applied the firewall approach to a large-scale banking system [190].

2.3.11 Cluster Identification

Laski and Szemer presented a test case selection technique based on analysis of the CFG of the program under test [115]. Their technique identifies single-entry, single-exit subgraphs of CFG called clusters. Given a program P and its modified version P',

- each cluster in P encapsulates some modifications to P,
- there is a unique cluster in P' that corresponds to the cluster in P, and
- when clusters in each graph are replaced by single nodes, there is a one-to-one correspondence between nodes in both graphs.

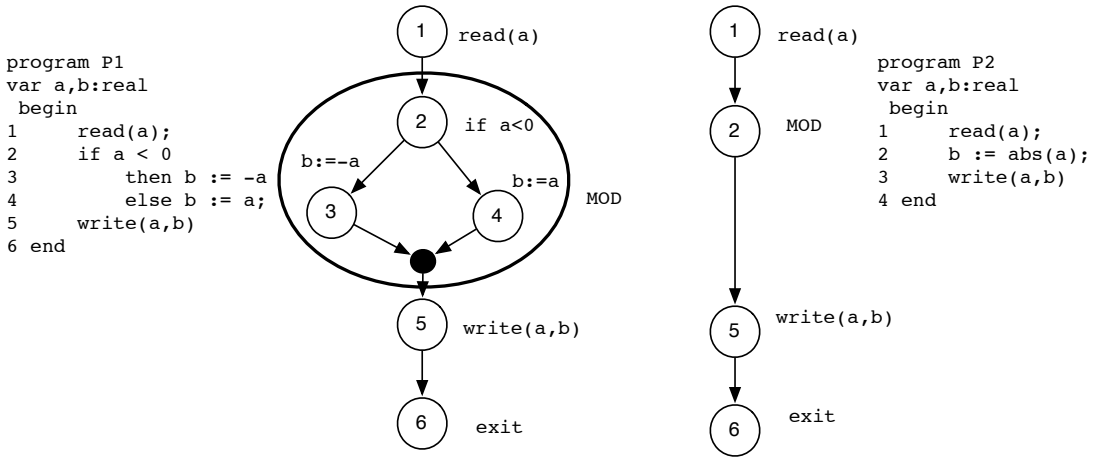

Figure 2.6: Example of isomorphism between reduced CFGs, taken from Laski and Szemer [115]. Program `P2` is a modified version of `P1`. Note that the modification made to `P2` is enclosed in `MOD` cluster.

The CFGs of the original program and the modified program are reduced using a set of operators such as node collapse and node removal. During the process, if the counterpart of a node from the CFG of the original program cannot be found in the CFG of the modified program, this node is labelled as 'MOD', indicating a modification at the node. Eventually, all the modifications will be enclosed in one or more MOD cluster nodes. As with other test case selection techniques, their technique requires that the tester records the execution history of each test case in the test suite. Once clustering is completed, test case selection is performed by selecting all the test cases for which the corresponding execution path enters any of the MOD clusters.

For example, consider Figure 2.6. Program `P2` is a modified version of `P1`. By collapsing nodes 2, 3, and 4 into a `MOD` cluster, the CFGs of two programs become isomorphic. Any test case that enters `MOD` with `P1` also enters `MOD` in `P2` and vice versa. Note that `MOD` in `P1` is considered to be single-entry/single-exit by the inclusion of a dummy merging node (the solid black node). The modification made to `P1` is enclosed in the `MOD` cluster. If a test case has an execution trace that enters `MOD`, it should be selected.

The strength of the cluster identification technique is that it guarantees to select all modification-traversing test cases regardless of the type of the modification, i.e. addition or deletion of statements and control structures. However, since the clusters can encapsulate much larger areas of the SUT than the scope of actual modification, the technique may also select test cases that are not modification-traversing. In this sense the approach sacrifices precision in order to achieve safety.

2.3.12 Design-based Approach

Briand et al. presented a black-box, design level regression test selection approach for UML-based designs [21, 22]. Assuming that there is traceability between the design and regression test cases, it is possible to perform regression test selection of code-level test cases from the impact analysis of UML design models. Briand et al. formalised possible changes in UML models, and classified the relevant test cases into the categories defined by Leung and White [121]: obsolete, retestable and reusable. They implemented an automated impact analysis tool for UML and empirically evaluated it using both student projects and industrial case studies.

The results showed that the changes made to a model can have a widely variable impact on the resulting system, which, in turn, yield varying degrees of reduction of effort in terms of the number of selected test cases. However, Briand et al. noted that the automated impact analysis itself can be valuable, especially for very large systems, such as the cruise control and monitoring system they studied. The UML use-cases of the model of the system had 323,614 corresponding test cases. UML-based models have been also considered by Dent et al. [39], Pilskalns et al. [156] and Farooq et al. [56] for regression test selection; Le Traon et al. [117] and Wu and Offutt [222] considered the use of UML models in the wider context of regression testing in general. Muccini et al. considered the RTS problem at the software architecture level, although they did not use UML for the representation [144, 145].

Test Case	Fault revealed by test case									
	1	2	3	4	5	6	7	8	9	10
A	x				x					
B	x				x	x	x			
C	x	x	x	x	x	x	x			
D					x					
E								x	x	x

Table 2.4: Example test suite with fault detection information, taken from Elbaum et al [53]. It is clearly beneficial to execute test case C first, followed by E.

2.4 Test Case Prioritisation

Test case prioritisation seeks to find the ideal ordering of test cases for testing, so that the tester obtains maximum benefit, even if the testing is prematurely halted at some arbitrary point. The approach was first mentioned by Wong et al. [220]; however, it was, in that work, only applied to test cases that were already selected by a test case selection technique. Subsequently, Harrold proposed the approach in more general context [78], which was empirically evaluated by Rothermel et al. [177].

For example, consider the test suite described in Table 2.4. Note that the example depicts an ideal situation in which fault detection information is known. The goal of prioritisation is to maximise early fault detection. It is obvious that the ordering A-B-C-D-E is inferior to B-A-C-D-E. In fact, any ordering that starts with the execution of C-E is superior to those that do not, because the subsequence C-E detects faults as early as possible; should testing be stopped prematurely, this ensures that the maximum possible fault coverage will have been achieved.

Note that the problem definition concerns neither versions of the program under test nor exact knowledge of modifications. Ideally, the test cases should be executed in the order that maximises early fault detection. However, fault detection information is typically not known until the testing is finished. In order to overcome the difficulty of knowing which tests reveal faults, test case prioritisation techniques depend on surro-

gates, hoping that early maximisation of a certain chosen surrogate property will result in maximisation of earlier fault detection. In a controlled regression testing environment, the result of prioritisation can be evaluated by executing test cases according to the produced ordering and measuring the fault detection rate.

2.4.1 Coverage-based Prioritisation

Structural coverage is a metric that is often used as the prioritisation criterion [49, 53, 54, 129, 163, 177, 178]. The intuition behind the idea is that early maximisation of structural coverage will also increase the chance of early maximisation of fault detection. Therefore, while the goal of test case prioritisation still remains as a higher fault detection rate, the prioritisation techniques actually aim to maximise early coverage.

Rothermel et al. reported empirical studies of several prioritisation techniques [177, 178]. They applied the same algorithm with different fault detection rate surrogates. The considered surrogates were: branch-total, branch-additional, statement-total, statement-additional, Fault Exposing Potential (FEP)-total, and FEP-additional.

The branch-total approach prioritises test cases according to the number of branches covered by individual test cases, while branch-additional prioritises test cases according to the additional number of branches covered by individual test cases. The statement-total and statement-additional approaches do the same thing with the number of program statements instead of branches. Algorithmically, 'total' approaches are essentially instances of greedy algorithms whereas 'additional' approaches are essentially instances of additional greedy algorithms.

The FEP of a test case is measured using program mutation. Program mutation introduces a simple syntactic modification to the program source, producing a mutant version of the program [27]. This mutant is said to be *killed* by a test case if the test case reveals the difference between the original program and the mutant. Given a set of mutants, the mutation score of a test case is the ratio of mutants that are killed

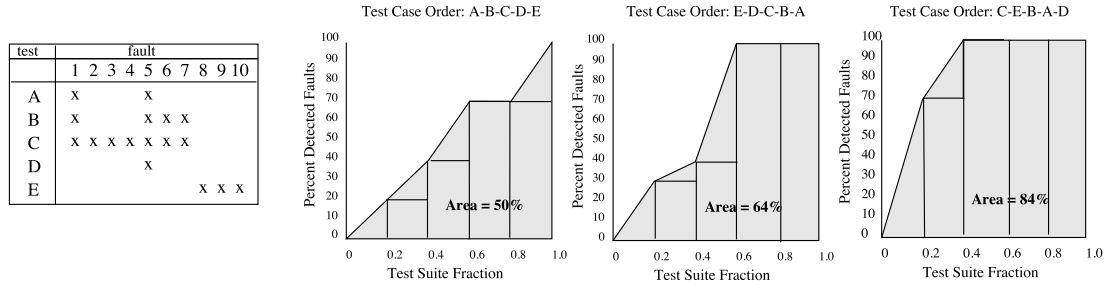

Figure 2.7: Average Percentage of Fault Detection : higher APFD values mean higher average fault detection rate [53].

by the test case to the total kill-able mutants. The FEP-total approach prioritises test cases according to the mutation score of individual test cases, while the FEP-additional approach prioritises test cases according to the additional increase in mutation score provided by individual test cases. Note that FEP criterion can be constructed to be at least as strong *strong* as structural coverage; to kill a mutant, a test case not only need to achieve the coverage of the location of mutation but also to execute the mutated part with a set of test input that can kill the mutant. In other words, coverage is necessary but not sufficient to kill the mutant.

It is important to note that all the 'additional' approaches may reach 100% realisation of the utilised surrogate before every test case is prioritised. For example, achieving 100% branch coverage may not require all the test cases in the test suite, in which case none of the remaining test cases can increase the branch coverage. Rothermel et al. reverted to the 'total' approach once such condition is met.

The results were evaluated using the Average Percentage of Fault Detection (APFD) metric. Higher APFD values denote faster fault detection rates. When plotting the percentage of detected faults against the number of executed test cases, APFD can be calculated as the area below the plotted line, as shown in Figure 2.7 taken from Elbaum

et al. [53]. Figure 2.7 shows the APFD values of orderings of the test suite described in Table 2.4. After the execution of the subsequence C-E, a 100% fault detection is achieved and the ordering of the remaining test cases does not affect the APFD value.

More formally, let T be the test suite containing n test cases and let F be the set of m faults revealed by T. For ordering T', let TF_i be the order of the first test case that reveals the ith fault. The APFD value for T' is calculated as following [51]:

$$APFD = 1 - \frac{TF_1 + \ldots + TF_m}{nm} + \frac{1}{2n}$$

Note that, while APFD is commonly used to evaluate test case prioritisation techniques, it is not the aim of test case prioritisation techniques to maximise APFD. Maximisation of APFD would be only possible when every fault that can be detected by the given test suite is already known. This would imply that all test cases have been already executed, which would nullify the need to prioritise. APFD is computed after the prioritisation only to evaluate the performance of the prioritisation technique.

Rothermel et al. compared the proposed prioritisation techniques to random prioritisation, optimal prioritisation, and no prioritisation, using the Siemens suite programs. Optimal prioritisation is possible because the experiment was performed in a controlled environment, i.e. the faults were already known. The results show that all the proposed techniques produce higher APFD values than random or no prioritisation. The surrogate with the highest APFD value differed between programs, suggesting that there is no single best surrogate. However, on average across the programs, the FEP-additional approach performed most efficiently, producing APFD value of 74.5% compared to the 88.5% of the optimal approach. It should still be noted that these results are dependent on many factors, including the types of faults used for evaluation and types of mutation used for FEP, limiting the scope for generalisation.

Elbaum et al. extended the empirical study of Rothermel et al. by including more

programs and prioritisation surrogates [53]. Among the newly introduced prioritisation surrogates, function-coverage and function-level FEP enabled Elbaum et al. to study the effects of granularity on prioritisation. Function-coverage of a test case is calculated by counting the number of functions that the test case executes. Function-level FEP is calculated, for each function f and each test case t, by summing the ratio of mutants in f killed by t. Elbaum et al. hypothesised that approaches with coarser granularity would produce lower APFD values, which was confirmed statistically.

Jones and Harrold applied the greedy-based prioritisation approach to Modified Condition/Decision Coverage (MC/DC) criterion [99]. MC/DC is a 'stricter form' of branch coverage; it requires execution coverage at condition level. A condition is a Boolean expression that cannot be factored into simpler Boolean expressions. By checking each condition in decision predicates, MC/DC examines whether each condition independently affects the outcome of the decision [32]. They presented an empirical study that contained only an execution time analysis of the prioritisation technique and not an evaluation based on fault detection rate.

Srivastava and Thiagarajan combined the greedy-based prioritisation approach with regression test selection [194]. They first identified the modified code blocks in the new version of the SUT by comparing its binary code to that of the previous version. Once the modified blocks are identified, test case prioritisation is performed using greedy-based prioritisation, but only with respect to the coverage of modified blocks.

Do and Rothermel applied coverage-based prioritisation techniques to the JUnit testing environment, a popular unit testing framework [40]. The results showed that prioritised execution of JUnit test cases improved the fault detection rate. One interesting finding was that the random prioritisation sometimes resulted in an APFD value higher than the *untreated* ordering, i.e. the order of creation. When executed in the order of creation, newer unit tests are executed later. However, it is the newer unit tests that have higher chance of detecting faults. It turns out that random prioritisation could exploit

Hardware	Operating System	Network Connection	Memory
Desktop	MS Windows	Dial-up	256MB
Laptop	Linux	DSL	512MB
Smartphone	Mac OS X	Cable	1GB

Table 2.5: Example testing environment factors

this weakness of untreated ordering in some cases.

Li et al. applied various meta-heuristics for test case prioritisation [126]. They compared random prioritisation, hill climbing algorithm, a genetic algorithm, a greedy algorithm, the additional greedy algorithm, and a two-optimal greedy algorithm. The greedy algorithm corresponds to the *total* approaches outlined above, whereas the additional greedy algorithm corresponds to the *additional* approaches outlined above. The two-optimal greedy is similar to the greedy algorithm except that it considers two candidates at the same time rather than a single candidate for the next order. They considered the Siemens suite programs and the program `space`, and evaluated each technique based on APBC (Average Percentage of Block Coverage) instead of APFD. The results showed that the additional greedy algorithm is the most efficient in general.

2.4.2 Interaction Testing

Interaction testing is required when the SUT involves multiple combinations of different components. For example, consider the testing environment described in Table 2.5. Each column represents a component that can have one of multiple possible settings. These are called *factors*. Each row in a column represents one possible setting. These are called *levels*. From the factors listed in Table 2.5, $3^4 = 81$ combinations arise. As the system grows larger, exhaustive testing of all possible combinations of factors requires exponentially more tests.

Instead of testing exhaustively, *pair-wise* interaction testing requires only that every individual pair of interactions between different factors are included at least once in

the testing process. In the example in Table 2.5, the number of combinations to test is reduced from 81 to 9. The reduction grows larger as more factors and levels are involved. More formally, the problem of obtaining interaction testing combinations can be expressed as the problem of obtaining a *covering array*, $CA(N; t, k, v)$, which is an array with N rows and k columns, v is the number of levels associated with each factor, and t is the *strength* of the interaction coverage (2 in the case of pair-wise interaction test).

Many approaches have been studied for the generation of interaction test suites, some of which share the same basic principles of test case prioritisation. For example, the greedy approach aims to find, one by one, the 'next' test case that will increase the k-way interaction coverage the most [25, 35], which resembles the greedy approach to test case prioritisation. However, the similarities are not just limited to the generation of interaction test suite. Bryce and Colbourn assume that testers may value certain interactions higher than others [23, 24]. For example, an operating system with a larger user base may be more important than one with smaller user base. After weighting each level value for each factor, they calculate the combined benefit of a given test by adding the weights of each level value selected for the test. They present a Deterministic Density Algorithm (DDA) that prioritises interaction tests according to their combined benefit. Qu et al. compared different weighting schemes used for prioritising covering arrays [158, 159].

Bryce and Memon also applied the principles of interaction coverage to the testing of Event-Driven Software (EDS) [26]. EDS takes sequences of events as input, changes state and output new event sequences. A common example would be GUI-based programs. Bryce and Memon interpreted t-way interaction coverage as sequences that contain different combinations of events over t unique GUI windows. Interaction coverage based prioritisation of test suites was compared to different prioritisation techniques such as unique event coverage (the aim is to cover as many unique events as possible, as early as

possible), longest to shortest (execute the test case with the longest event sequence first) and shortest to longest (execute the test case with the shortest event sequence first). The empirical evaluation showed that interaction coverage based testing of EDS can be more efficient than the other techniques, provided that the original test suite contains higher interaction coverage. Note that Bryce and Memon did not try to generate additional test cases to improve interaction coverage; they only considered permutations of existing test cases.

2.4.3 Prioritisation Approaches Based on Other Criteria

While the majority of existing prioritisation literature concerns structural coverage in some form or other, there are prioritisation techniques based on other criteria [120, 193, 202, 226].

Distribution-based Approach Leon and Podgurski introduced distribution-based filtering and prioritisation [120]. Distribution-based techniques minimise and prioritise test cases based on the distribution of the profiles of test cases in the multi-dimensional profile space. Test case profiles are produced by the dissimilarity metric, a function that produces a real number representing the degree of dissimilarity between two input profiles. Using this metric, test cases can be clustered according to their similarities. The clustering can reveal some interesting information. For example:

- Clusters of similar profiles may indicate a group of redundant test cases
- Isolated clusters may contain test cases inducing unusual conditions that are perhaps more likely to cause failures
- Low density regions of the profile space may indicate uncommon usage behaviours

The first point is related to reduction of effort; if test cases in a cluster are indeed very similar, it may be sufficient to execute only one of them. The second and third

point are related to fault-proneness. Unusual conditions and uncommon behaviours are by definition harder to reproduce than more common conditions and behaviours. Therefore, the corresponding parts of the program are likely to be tested less than other, more frequently used parts of the program. Assigning a high priority to test cases that execute these unusual behaviours may increase the chance of early fault detection. A good example might be exception handling code.

Leon and Podgurski developed new prioritisation techniques that combine coverage-based prioritisation with distribution-based prioritisation. This hybrid approach is based on the observation that *basic coverage maximisation* performs reasonably well compared to *repeated coverage maximisation*. Repeated coverage maximisation refers to the prioritisation technique of Elbaum et al. [53], which, after realising 100% coverage, repeatedly prioritises test cases starting from 0% coverage again. In contrast, basic coverage maximisation stops prioritising when 100% coverage is achieved. Leon and Podgurski observed that the fault detection rate of repeated coverage maximisation is not as high as that of basic coverage maximisation. This motivated them to consider a hybrid approach that first prioritises test cases based on coverage, then switches to distribution-based prioritisation once the basic coverage maximisation is achieved. They considered two different distribution-based techniques. The one-per-cluster approach samples one test case from each cluster, and prioritises them according to the order of cluster creation during the clustering. The failure-pursuit approach behaves similarly, but it adds the k closest neighbours of any test case that finds a fault. The results showed that the distribution-based prioritisation techniques could outperform repeated coverage maximisation.

Human-based Approach Tonella et al. combined Case-Based Reasoning (CBR) with test case prioritisation [202]. They utilised a machine learning technique called *boosting*, which is a framework to combine simple learners into a single, more general and effective learner [65]. They adopted a boosting algorithm for ranking learning called

Rankboost [66]. The algorithm takes a test suite, T, an initial prioritisation index, f, and a set of pair-wise priority relations between test cases, Φ, as input. The pair-wise priority relation is obtained from comparisons of test cases made by the human tester. The output is a ranking function $H : T \to \mathbb{R}$ such that, with test cases t_1 and t_2, $t_1 \prec t_2$ if $H(t_1) > H(t_2)$. The ranking function H is then used to prioritise test cases.

They used the statement coverage metric and the cyclomatic complexity computed for the functions executed by test cases as the initial prioritisation index. The test suite of the `space` program was considered. In order to measure the human effort required for the learning process, different test suite sizes were adopted, ranging from 10 to 100 test cases. The results were compared to other prioritisation techniques including optimal ordering, random prioritisation, statement coverage prioritisation, and additional statement coverage prioritisation (the latter two correspond to statement-total and statement-additional respectively).

The results showed that, for all test suite sizes, the CBR approach was outperformed only by the optimal ordering. The number of pair-wise relations entered manually showed a linear growth against the size of test suites. Tonella et al. reported that, for test suites of `space` with fewer than 60 test cases, the CBR approach can be more efficient than other prioritisation techniques with limited human effort. Note that empirical evaluation was performed based on an *ideal* user model, i.e. it was assumed that the human tester always makes the correct decision when comparing test cases. One notable weakness of this approach was that it did not scale well. The input from the human tester becomes inconsistent beyond a certain number of comparisons, which in turn limits the size of the learning samples for CBR.

This thesis introduces the use of clustering techniques to make the human-based prioritisation approach more scalable. For more details, refer to Chapter 6.

Probabilistic Approach Kim and Porter proposed a history based approach to priori-

tise test cases that are already selected by regression test selection [102]. If the number of test cases selected by an RTS technique is still too large, or if the execution costs are too high, then the selected test cases may have to be further prioritised. Since the relevance to the recent change in SUT is assumed by the use of an RTS technique, Kim et al. focus on the execution history of each test case, borrowing from statistical quality control. They define the probabilities of each test case tc to be selected at time t as $P_{tc,t}(H_{tc}, \alpha)$, where H_{tc} is a set of t timed observations $\{h_1, \ldots, h_t\}$ drawn from previous runs of tc and α is a smoothing constant. Then the probability $P_{tc,t}(H_{tc}, \alpha)$ is defined as follows:

$$P_0 = h_1$$
$$P_k = \alpha h_k + (1 - \alpha)P_{k-1} \quad (0 \leq \alpha \leq 1, k \geq 1)$$

Different definitions of H_{tc} result in different prioritisation approaches. For example, Kim et al. define Least Recently Used (LRU) prioritisation by using test case execution history as H_{tc} with α value that is as close to 0 as possible. The empirical evaluation showed that the LRU prioritisation approach can be competitive in a severely constrained testing environment, i.e. when it is not possible to execute all test cases selected by an RTS technique.

Mirarab and Tahvildari took a different probabilistic approach to test case prioritisation using Bayesian Networks [142]. The Bayesian Network model is built upon changes in program elements, fault proneness of program elements and probability of each test case to detect faults. Mirarab and Tahvildari extended the approach by adding a feedback route to update the Bayesian Network as prioritisation progresses [143]. For example, if a test case successfully detects a fault, there is a decrease in the probability for other test cases to be selected in order to cover the same program element.

History-based Approach Sherriff et al. presented a prioritisation technique based on association clusters of software artefacts obtained by a matrix analysis called Singular Value Decomposition (SVD) [189]. The prioritisation approach depends on three elements: association clusters, relationship between test cases and files and a modification vector. Association clusters are generated from a change matrix using SVD; if two files are often modified together as a part of a bug fix, they will be clustered into the same association cluster. Each file is also associated with test cases that affect or execute it. Finally, a new system modification is represented as a vector in which the value indicates whether a specific file has been modified. Using the association clusters and the modification vector, it is then possible to assign each file with a priority that corresponds to how closely the new modification matches each test case. One novel aspect of this approach is that any software artefact can be considered for prioritisation. Sherriff et al. noted that the faults that are found in non-source files, such as media files or documentation, can be as severe as those found in source code.

Requirement-based Approach Srikanth et al. presented requirement-based test case prioritisation [193]. Test cases are mapped to software requirements that are tested by them, and then prioritised by various properties of the mapped requirements, including customer-assigned priority and implementation complexity. One potential weakness of this approach is the fact that requirement properties are often estimated and subjective values. Krishnamoorthi and Sahaaya developed a similar approach with additional metrics [112].

Model-based Approach Korel et al. introduced a model based prioritisation approach [106, 108, 110]. Their initial approach was called *selective* prioritisation, which was strongly connected to RTS [108]. Test cases were classified into a high priority set, TS_H, and a low priority set, TS_L. They defined and compared different definitions of

high and low priority test case, but essentially a test case is assigned high priority if it is relevant to the modification made to the model. The initial selective prioritisation process consists of the random prioritisation of TS_H followed by the random prioritisation of TS_L. Korel et al. developed more sophisticated heuristics based on the dependence analysis of the models [106, 110].

Other Approaches The use of mutation score for test case prioritisation has been analysed by Rothermel et al. along with other structural coverage criteria [177, 178]. Hou et al. considered interface-contract mutation for the regression testing of component-based software and evaluated it with the *additional* prioritisation technique [90].

Sampath et al. presented the prioritisation of test cases for web applications [187]. The test cases are, in this case, recorded user sessions from the previous version of the SUT. Session-based test cases are thought to be ideal for testing web applications because they tend to reflect the actual usage patterns of real users, thereby making for realistic test cases. They compared different criteria for prioritisation such as the number of HTTP requests per test case, coverage of parameter values, frequency of visits for the pages recorded in sessions and the number of parameter values. The empirical evaluations showed that prioritised test suites performed better than randomly ordered test suites, but also that there is not a single prioritisation criterion that is always best. However, the 2-way parameter-value criterion, the prioritisation criterion that orders tests to cover all pair-wise combinations of paramenter-values between pages as soon as possible, showed the highest APFD value for 2 out of 3 web applications that were studied.

Fraser and Wotawa introduced a model-based prioritisation approach [64]. Their prioritisation technique is based on the concept of *property relevance* [63]. A test case is relevant to a model property if it is theoretically possible for the test case to violate the property. The relevance relation is obtained by the use of a model-checker, which is used as the input to the greedy algorithm. While they showed that property-bsed

prioritisation can outperform coverage-based prioritisation, they noted that the performance of property-based prioritisation is heavily dependent on the quality of the model spicification.

A few techniques and analyses used for test suite minimisation or regression test selection problem have been also applied to test case prioritisation. Rummel et al. introduced a prioritisation technique based on data-flow analysis by treating each *du* pair as a testing requirement to be covered [179]. Smith et al. introduced a prioritisation technique based on a call-tree model, which they also used for test suite minimisation [191]. They prioritised test cases according to the number of call-tree paths covered by each test case. Jeffrey and Gupta prioritised test cases using relevant slices [96], which was also used for regression test selection [4]. Each test case was associated with output statements, from which relevant slices were calculated. Then test cases were prioritised according to the sum of two elements: the size of the corresponding relevant slice and the number of statements that are executed by the test case but do not belong to the relevant slice. Both elements were considered to correlate to the chance of revealing a fault introduced by a recent change.

2.4.4 Cost-Aware Test Case Prioritisation

Unlike test suite minimisation and regression test selection, the basic definition of test case prioritisation does not involve filtering out test cases, i.e. it is assumed that the tester executes the entire test suite following the order given by the prioritisation technique. This may not be feasible in practice due to limited resources. A number of prioritisation techniques addressed this problem of the need to be cost-aware [42, 54, 210, 225].

With respect to cost-awareness, the basic APFD metric has two limitations. First, it considers all faults to be equally severe. Second, it assumes that every test case costs the same in resources. Elbaum et al. extended the basic APFD metric to APFD_c so that the metric incorporates not just the rate of fault detection but also the severity of detected

	Fault										
ID	1	2	3	4	5	6	7	8	9	10	
Severity	1	1	1	1	1	1	1	1	3	3	
Testcase											Time
A	x				x						1
B	x				x	x	x				1
C	x	x	x	x	x	x	x				6
D					x						1
E								x	x	x	1

Table 2.6: Example test suite with execution time and fault severity information. The original test suite is taken from Elbaum et al. [53]. Note that the test case C detects most faults at the cost of taking 6 times longer than other test cases. Also note that the faults detected by the test case E have higher severity values than the others. This leads to different prioritisation to a non cost-cognizant approach.

faults and the expense of executing test cases [54]. Whereas the x-axis and y-axis in Figure 2.7 show the number of executed test cases and the number of detected faults respectively, they denote the sum of execution costs of test cases and the sum of fault severity values of detected faults in an APFD_c plot. An ordering of test cases according to the APFD_c metric detects severer faults earlier.

More formally, let T be the set of n test cases with costs $t_1, \ldots, t_n$, and let F be the set of m faults with severity values $f_1, \ldots, f_m$. For ordering T', let TF_i be the order of the first test case that reveals the ith fault. APFD_c of T' is calculated as following:

$$APFD_c = \frac{\sum_{i=1}^{m} (f_i \times (\sum_{j=TF_i}^{n} t_j - \frac{1}{2} t_{TF_i}))}{\sum_{i=1}^{n} t_i \times \sum_{i=1}^{m} f_i}$$

For example, consider the test suite described in Table 2.6. The fault detection information is the same as one shown in Table 2.4, but it additionally contains the fault severity information and test case execution time. Figure 2.8 shows how APFD_c is calculated for the orderings of C-E-A-B-D and E-C-A-B-D from the test suite in Table 2.6. The APFD_c value for E-C-A-B-D is 77.5%, which is higher than the APFD_c value of C-E-A-B-D, 59.5%. Note that C-E-A-B-D is deemed to be better than E-C-A-B-D by

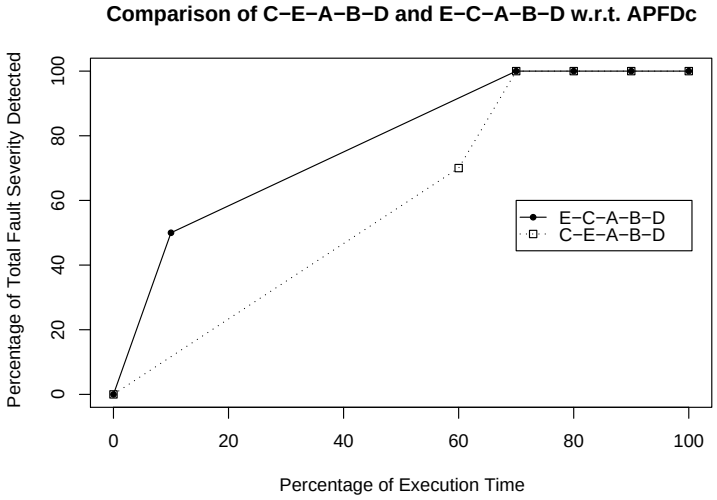

Figure 2.8: $APFD_c$ plot of C-E-A-B-D and E-C-A-B-D from the test suite shown in Table 2.6. C detects 7 faults, but collectively they are not as severe as the 3 faults detected by E. E also takes shorter time to execute than C.

the non cost-cognizant approach in Figure 2.7 (Section 2.4.1). However, using the cost-cognizant approach, E-C-A-B-D is favoured over C-E-A-B-D.

Elbaum et al. applied random ordering, additional statement coverage prioritisation, additional function coverage prioritisation and additional fault index prioritisation techniques to `space`, which contains faults discovered during the development stage. They adopted several different models of test case cost and fault severity, including uniform values, random values, normally distributed values and models taken from the Mozilla open source project. The empirical results achieved by synthetically adding cost severity models to `space`. This enabled them to observe the impact of different severity and cost models. They claimed two practical implications. With respect to test case cost, they proposed the use of many small test cases rather than a few large test cases. Clearly the number of possible prioritisations is higher with a test suite that contains many small test cases, compared to one with a small number of large test cases. It was also claimed that having different models of fault severity distribution can also impact the

efficiency of testing. This is only true when the prioritisation technique considers the fault detection history of previous tests.

Elbaum et al. compared two different severity distribution models: linear and exponential. In the linear model, the severity values grow linearly as the severity of faults increase, whereas they grow exponentially in the exponential model. If the previous fault detection history correlates to the fault detection capability of the current iteration of testing, the exponential model ensures that test cases with a history of detecting severer faults are executed earlier.

Walcott et al. presented a time-aware prioritisation technique [210]. Time-aware prioritisation does not prioritise the entire test suite; it aims to produce a subset of test cases that are prioritised and can be executed within the given time budget. More formally, it is defined as following:

Given: A test suite, T, the collection of all permutations of elements of the power set of permutations of T, $perms(2^T)$, the time budget, t_{max}, a time function $time : perms(2^T) \rightarrow \mathbb{R}$, and a fitness function $fit : perms(2^T) \rightarrow \mathbb{R}$:

Problem: Find the test tuple $\sigma_{max} \in perms(2^T)$ such that $time(\sigma_{max}) \leq t_{max}$ and $\forall \sigma' \in perms(2^T)$ where $\sigma_{max} \neq \sigma'$ and $time(\sigma') \leq t_{max}$, $fit(\sigma_{max}) > fit(\sigma')$.

ß

Intuitively, a time-aware prioritisation technique selects and prioritises test cases at the same time so that the produced ordered subset yields higher fault detection rates within the given time budget. They utilised a genetic algorithm, combining selection and prioritisation into a single fitness function. The selection component of the fitness function is given higher weighting so that it dominates the overall fitness value produced. The results of the genetic algorithm were compared to random ordering, reverse ordering

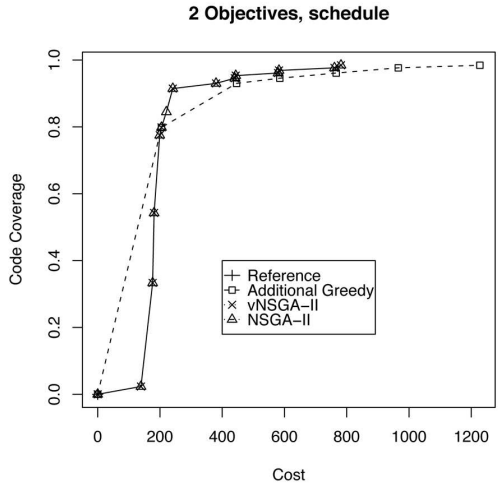

Figure 2.9: Comparison of different heuristics to the reference Pareto-frontier obtained from the test suite of `schedule` using exhaustive search, taken from Yoo and Harman [225]. The solid line represents the trade-off between code coverage and cost of test case execution.

and the optimal ordering. The results showed that time-aware prioritisation produces higher rates of fault detection compared to random, initial, and reverse ordering. However, they did not compare the time-aware prioritisation to the existing, non time-aware prioritisation techniques. Note that non time-aware prioritisation techniques can also be executed in 'time-aware' manner by stopping the test when the given time budget is exhausted.

While Yoo and Harman studied test suite minimisation [225], their multi-objective optimisation approach is also relevant to the cross-cutting concern of cost-awareness. By using multi-objective optimisation heuristics, they obtained a Pareto-frontier which represents the trade-offs between the different criteria including cost. Figure 2.9 is such an example obtained from the test suite of the Siemens suite program, `schedule`. Two observations are of interest. First, the reference Pareto-frontier obtained from exhaustive search reveals that full code coverage can be achieved with lower cost compared to greedy

prioritisation. Second, the reference Pareto-frontier contains more decision points than that produced by greedy prioritisation. When there is a constraint on cost, the knowledge of Pareto-frontier can therefore provide the tester with more information to achieve higher coverage. The tester can then prioritise the subset selected by observing the Pareto-frontier.

The cost-constraint problem has also been analysed using Integer Linear Programming (ILP) [91, 230]. Hou et al. considered the cost-constraint in web service testing [91]. Users of web services are typically assigned with a usage quota; testing a system that uses web services, therefore, has to consider the remaining quota for each web service. The ILP approach was later analysed in more generic context using execution time of each test as cost factor [230].

Do and Rothermel studied the impact of time constraints on the cost-effectiveness of existing prioritisation techniques [42]. In total six different prioritisation approaches were evaluated: original order, random order, total block coverage, additional block coverage, Bayesian Network approach without feedback, Bayesian Network approach with feedback. They considered four different time constraints, each of which allows $\{25\%, 50\%, 75\%, 100\%\}$ of time required for the execution of all test cases. Each prioritisation approach was evaluated under these constraints using a cost-benefit model. The results showed that, although time constraints affect techniques differently, it is always beneficial to adopt some prioritisation when under time constraints. The original ordering was always affected the most severely.

2.5 Meta-Empirical Studies

Recently, the meta-empirical study of regression testing techniques has emerged as a separate subject in its own right. It addresses cross-cutting concerns such as cost-benefit analysis of regression testing techniques and the studies of evaluation methodology for

these techniques. Both studies seek to provide more confidence in efficiency and effectiveness of regression testing techniques. Work in these directions are still in early stages compared to the bodies of work available for minimisation, selection or prioritisation techniques. However, these studies are expected to make significant contributions towards the technology transfer.

Empirical evaluation of any regression testing technique is inherently a *post-hoc* process that assumes the knowledge of a set of known faults. Without the *a-priori* knowledge of faults, it would not be possible to perform a controlled experiment of comparing different regression testing techniques. This poses a challenge to the empirical evaluation of techniques, since the availability of fault data tends to be limited [5].

Andrews et al. performed an extensive comparison between real faults and those seeded by mutation [5]. One concern when using mutation faults instead of real faults is that there is no guarantee that the detection of mutation faults can be an accurate predictor of the detection of real faults. After considering various statistical data such as the ratio and distribution of fault detection, Andrews et al. concluded that mutation faults can indeed provide a good indication of the fault detection capability of the test suite, assuming that mutation operators are carefully selected and equivalent mutants are removed. However, they also note that, while mutation faults were not easier to detect than real faults, they were also not harder to detect. Do and Rothermel extended this study by focusing the comparison on the result of test case prioritisation techniques [43, 45]. Here, they considered whether evaluating prioritisation techniques against mutation faults produces results different from evaluating against hand seeded faults. Based on the comparison of these two evaluation methods, it was concluded that mutation faults can be safely used in place of real or hand-seeded faults.

Although it was not their main aim, Korel et al. made an important contribution to the empirical evaluation methodology of regression testing techniques through the empirical evaluation of their prioritisation techniques [106, 108, 110]. They noted that,

in order to compare different prioritisation techniques in terms of their rate of fault detection, they need to be evaluated using all possible prioritised sequences of test cases that may be generated by each technique. Even deterministic prioritisation algorithms, such as the greedy algorithm, can produce different results for the same test suite if some external factors change; for example, if the ordering of the initial test suite changes, there is a chance that the greedy algorithm will produce a different prioritisation result. Korel et al. argued, therefore, that the rate of fault detection should be measured in average across all possible prioritised sequences. They introduced a new metric called Most Likely average Position (MLP), which measures the average relative position of the first test case that detects a specific fault.

Elbaum et al. extended the empirical studies of prioritisation techniques with the Siemens suite and `space` [53] by performing statistical analysis of the variance in APFD [49]. The APFD values were analysed against various program, change, and test metrics. Program metrics included mean number of executable statements, mean function size across all functions, etc. Change metrics included number of functions with at least one changed statement, number of statements inserted or deleted, etc. Test metrics included number of tests in the test suite, percentage of tests reaching a changed function, mean number of changed functions executed by a test over a test suite, etc. The aim was to identify the source of variations in results. They reported that the metrics that reflected normalised program characteristics (such as mean function size across the program) and characteristics of test suites in relation to programs (such as mean percentage of functions executed by a test over a test suite) were the primary contributors to the variances in prioritisation. While they reported that this finding was not the anticipated one, it showed that the prioritisation results are the product of closely coupled interactions between programs under test, changes and test suites.

Empirical evaluation of different techniques can benefit from a shared evaluation framework. Rothermel and Harrold presented a comparison framework for RTS tech-

niques [169], which was used to compare different RTS techniques [171]. While minimisation and prioritisation techniques lack such a framework, certain metrics have been used as a *de facto* standard evaluation framework. Rate of reduction in size and rate of reduction in fault detection capability have been widely used to evaluate test suite minimisation techniques [18, 77, 94, 95, 133, 134, 176, 191, 198, 220, 221]. Similarly, Average Percentage of Fault Detection (APFD) [53] has been widely used to evaluate prioritisation techniques [40, 43, 45, 49–51, 53, 64, 91, 96, 120, 142, 143, 163, 164, 178, 179, 202, 210, 226, 230].

Rothermel et al. studied the impact of test suite granularity and test input grouping on the cost-effectiveness of regression testing [163, 164]. They first introduced the concept of *test grains*, which is the smallest unit of test input that is executable and checkable. Test cases are constructed by grouping test grains. Based on this, they defined test suite granularity as the number of test grains in a test case, and test input grouping as the way test grains are added to each test case, e.g. randomly or grouped by their functionality. They reported that having a coarse grained test suite did not significantly compromise the fault detection capability of the test suite, but resulted in decreased total execution time. The savings in execution time can be explained by the fact that a coarse grained test suite contains fewer test cases, thereby reducing the set-up time and other overheads that occur between execution of different test cases. However, they did not consider the cost of the test oracle. It is not immediately obvious whether the cost of a test oracle would increase or decrease as the test suite granularity increases. This oracle cost could affect the overall cost-effectiveness.

Kim et al. studied the impact of test application frequency to the cost-effectiveness of RTS techniques [103, 104]. Their empirical studies showed that the frequency of regression test application has a significant impact on cost-effectiveness of RTS techniques. They reported that RTS techniques tend to be more cost-effective when the frequency of test application is high. It implies that only small amount of changes are made be-

tween tests, which makes RTS more effective. However, as intervals between tests grows, changes are accumulated and RTS techniques tend to select more and more test cases, resulting in low cost-effectiveness. One interesting finding is that, as intervals between tests grows, random re-testing tends to work very well. With small testing intervals, the random approach fails to focus on the modification. As testing intervals increase, more parts of SUT need to be re-tested, improving the effectiveness of the random approach. Elbaum et al. studied the impacts of changes in terms of the quantitative nature of modifications [50]. They investigated how the cost-effectiveness of selection and prioritisation techniques is affected by various change metrics such as percentage of changed lines of code, average number of lines of code changed per function, etc. Their empirical analysis confirmed that the differences in these metrics can make a significant impact to the cost-effectiveness of techniques. However, they also reported that simple size of change, measured in lines of code, was not a predominant factor in determining the cost-effectiveness of techniques. Rather, it was the distribution of changes and the ability of test cases to reach these changes.

Elbaum et al. also presented a technique for selecting the most cost-effective prioritisation technique [52]. They applied a set of prioritisation techniques to the same set of programs, and analysed the resulting APFD metric values. Different techniques perform best for different programs; they applied the classification tree technique to predict the best-suited technique for a program. Note that the term 'cost-effectiveness' in this work means the efficiency of a prioritisation technique measured by the APFD metric; the computational cost of applying these techniques was not considered.

Rosenblum and Weyuker introduced a coverage-based cost-effective predictor for RTS techniques [162]. Their analysis is based on the coverage relation between test cases and program entities. If each program entity has a uniformly distributed probability of being changed in the next version, it is possible to predict the average number of test cases to be selected by a safe RTS technique using coverage relation information. They

evaluated their predictor with the TestTube RTS tool [31], using multiple versions of the `KornShell` [19] and an I/O library for Unix, `SFIO` [111], as subjects. Their predictor was reasonably accurate; for example, it predicted an average of 87.3% of the test suite to be selected for `KornShell`, when TestTube selected 88.1%. However, according to the cost model of Leung and White [124], the cost of coverage analysis for RTS per test case was greater than the cost of execution per test case, indicating that TestTube was not cost-effective. Harrold et al. introduced an improved version of the cost-effective predictor of Rosenblum et al. for more accurate cost-effectiveness prediction of version-specific RTS [80]. They evaluated their predictor using TestTube and another RTS tool, DejaVu [173].

Modelling the cost-effectiveness of regression testing techniques has emerged as an essential research topic as any analysis of cost-effectiveness should depend on some model. Leung and White introduced an early cost-model for regression testing strategies and compared the cost models of the *retest-all* strategy and the selective retesting strategy [124]. Malishevsky et al. presented detailed models of cost-benefit trade-offs for regression testing techniques [129]. They applied their models to the regression testing of `bash`, a popular Unix shell [160], with different ratio values of $\frac{f}{e+c}$, where f is the cost of omitting one fault, e is the additional cost per test and c is the result-validation cost per test. The results implied that if a regression testing technique does not consider f, it may overestimate the cost-effectiveness of a given technique. The cost model of Malichevsky et al. has been extended and evaluated against the prioritisation of JUnit test cases [47]. Smith and Kapfhammer studied the impact of the incorporation of cost into test suite minimisation [192]. Existing minimisation heuristics including HGS [77], delayed greedy [198] and 2-optimal greedy algorithm [126] were extended to incorporate the execution cost of each test case. Do and Rothermel considered the impact of time constraints on selection and prioritisation techniques across multiple consecutive versions of subject programs to incorporate software life-cycle factors into the study [44].

Reflecting the complexity of regression testing process, cost-effectiveness models often need to be sophisticated in order to incorporate multiple variables [44, 47, 129, 192]. However, complexity can be a barrier to uptake. Do and Rothermel introduced an approach based on statistical sensitivity analysis to simplify complicated cost models [46]. Their approach fixed certain cost factors that were deemed to be the least significant by the sensitivity analysis. The empirical evaluation showed that, while certain levels of simplification can still preserve the accuracy of the model, over-simplification may be risky.

Chapter 3

Multi-Objective Test Case Management

3.1 Introduction

In real world testing, there are often multiple test criteria. For example, different types of testing, such as functional testing and structural testing, require different testing criteria [77]. There also can be cases where it is beneficial for the tester to consider multiple test criteria because the single most ideal test criterion is simply unobtainable. For example, testers face the problem that real fault detection information cannot be known until regression testing is actually finished. Code coverage is one possible surrogate test adequacy criterion that is used in place of fault detection, but it is not the only one. Because one cannot be certain of a link between code coverage and fault detection it would be natural to supplement coverage with other test criteria, for example, past fault detection history.

Of course, the quality of the test data is not the only concern. Cost is also one of the essential criteria, because the whole purpose of test case selection and prioritisation is to achieve more efficient testing in terms of the cost. One important cost driver, considered

by other researchers [130, 210] is the execution time of the test suite.

In order to provide automated support for the selection of regression test data it therefore seems appropriate that a multi-objective approach is required that is capable of taking into account the subtleties inherent in balancing many, possibly competing and conflicting objectives. Existing approaches to regression test case selection (and prioritisation) have been single objective approaches that have sought to optimise a single objective function.

There has been recent work on a two objective formulation of test case prioritisation problem [130], that takes account of coverage and cost, using a single objective of coverage per unit cost. However, this approach conflates the two objectives into a single objective. Where there are multiple competing and conflicting objectives the optimisation literature recommends the consideration of a Pareto optimal optimisation approach [36, 196]. Such a Pareto optimal approach is able to take account of the need to balance the conflicting objectives, all of which the software engineer seeks to optimise.

This chapter[1] presents the multi-objective formulation of the test suite minimisation problem, showing how multiple objectives can be optimised using a Pareto efficient approach. Such an approach is believed to be well suited to test case management problem, because it is likely that a tester will want to optimise several possible conflicting constraints. The multi-objective formulation of test suite minimisation is instantiated in two empirical studies.

The rest of the chapter is organised as follows. Section 3.2 presents the existing single-objective paradigm. Section 3.3 introduces the multi-objective formulation of

[1]This chapter is an extended version of the author's ISSTA paper: S. Yoo and M. Harman, Pareto-Efficient Multi-Objective Test Case Selection. *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA)*, London, UK, pages 140-150, 2007.

3.2 Single Objective Paradigm

3.2.1 Test Suite Minimisation

The aim of test suite minimisation is to reduce the number of test cases that are required to achieve the given set of test requirements. More formally,

Test Suite Minimisation

Given: a test suite $T = \{t_1, t_2, \ldots, t_n\}$ and a set of test requirements $R = \{r_1, r_2, \ldots, t_m\}$

Problem: to find the smallest T' such that $T' \subset T$, $\forall r \in R(T'$ satisfies $r)$.

One possible candidate for the test requirements R is structural coverage. However, this formulation does not consider the cost of execution for the selected subset. In cost-cognizant formulation of test suite minimisation, each test case has associated cost of execution. The goal is to reduce test suite T to T' that satisfies the test requirements R while minimising the total cost of T'. This is essentially a weighted set cover problem:

Weighted Set Cover Problem

Given: a universe $\mathcal{U}$ with n elements, a set $\mathcal{S}$ of m subsets of $\mathcal{U}$ with $cost_1, \ldots, cost_m$

Problem: to find $\mathcal{C}$ such that $\mathcal{C} \subseteq \mathcal{S}$, $\bigcup_{S_i \in \mathcal{C}} S_i = \mathcal{U}$, and $\forall S' \subseteq \mathcal{S}[\bigcup_{S'_i \in \mathcal{S}'} S'_i = \mathcal{U} \rightarrow \sum_{S_i \in \mathcal{C}} cost_i \leq \sum_{S'_i \in S'} cost_i]$

Both the non-weighted and weighted version of set cover problem is known to be NP-hard [69]. Previous work did not consider the cost for test suite minimisation, and relied on heuristics for the non-weighted set cover problem [77, 89]. This thesis considers the use of greedy algorithm, which is known to have an approximation level of $ln(n)$ for weighted set cover problem.

3.2.2 Greedy Algorithm & Approximation Level

Algorithm 1: Outline of additional greedy algorithm

```
AdditionalGreedy(U, S)
(1)   C ← φ // covered elements in U
(2)   repeat
(3)     j ← min_k(cost_k/|S_k − C|)
(4)     add S_j to solution
(5)     C = C ⋃ S_j
(6)   until C = U
```

The additional greedy algorithm is illustrated in Algorithm 1. Let $\mathcal{U}$ be the universe, $\{e_1, \ldots, e_n\}$; $\mathcal{S}$ the set containing $S_1, \ldots, S_m$, the subsets of $\mathcal{U}$ such that $\bigcup_i S_i = \mathcal{U}$; $cost_1, \ldots, cost_m$ the cost of each subset in $\mathcal{S}$. Without loss of generality, it is assumed that there exists a subset $S' \subset \mathcal{S}$ that covers $\mathcal{U}$ completely. Through line (4) of Algorithm 1, the additional greedy algorithm picks $S_j \in \mathcal{S}$ according to the density of the set, $cost_j/|S_j - C|$. The minimum density corresponds to the maximum *increase in coverage* per *cost* in each iteration.

It is possible to show that results from the additional greedy algorithm are within $\ln n$ of the optimal cost [97].

Theorem 1 (Additional Greedy Approximation)**.**
Total cost of the maximum coverage obtained by additional greedy algorithm is within $\ln n$ of the optimal cost, OPT.

Proof. Let us assume that the elements in $\mathcal{U}$ are covered in order of $\{e_1, e_2, \ldots, e_n\}$, with ties resolved arbitrarily. Let h_j be the index of the set picked in iteration j, and C_j be the set of elements covered in iterations 1 to j.

Now, in jth iteration, S_{h_j} will cover a set of elements, $S'_{h_j} = S_{h_j} - C_{j-1}$. Now, let us define $p(e)$ as follows:

$$\forall e \in S'_{h_j}, p(e) = cost_{h_j}/|S'_{h_j}|$$

From the definition, it follows that:

$$\sum_{e \in \mathcal{U}} p(e) = \textit{cost of all elements covered}$$

Now, let e_i be covered in iteration j, which implies that $\{e_i, e_{i+1}, \ldots, e_n\}$ does not belong to C_{j-1}. Since OPT is the optimal cost for the maximum coverage, there is a solution of cost OPT that covers these elements, implying that there exists a set with density of $OPT/(n-i+1)$. Therefore it follows that:

$$cost_{h_j}/|S'_{h_j}| \leq OPT/(n-i+1)$$

which leads us to:

$$p(e_i) \leq OPT/(n-i+1)$$

From $p(e_i) \leq OPT/(n-i+1)$, it can be concluded that:

$$\sum_{e \in \mathcal{U}} p(e) \leq OPT \cdot (\frac{1}{n} + \frac{1}{n-1} + \ldots + 1) \leq OPT \cdot H_n \leq OPT \ln n$$

Therefore the total cost of the solution obtained by the additional greedy algorithm is within $\ln n$ of the optimal cost. □

The theoretical analysis that the additional greedy algorithm shows that the algorithm can produce a good approximation for the 2-objective test suite minimisation presented in Section 3.2. However, Section 3.3 also illustrates that there is a room for improvement between the optimal Pareto frontier of multi-objective test suite minimisation and the approximation obtained by the greedy approach.

3.3 Multi Objective Paradigm

This section introduces the multi-objective formulation of test case selection. Section 3.3.1 introduces the Pareto optimal formulation of the test case selection problem. Section 3.3.2 explores the theoretical properties of the two objective greedy algorithm, while section 3.3.3 shows the relationship between multi-objective selection and prioritisation.

3.3.1 Pareto Optimality

Pareto optimality is a notion from economics with broad range of applications in game theory and engineering [67]. The original presentation of the Pareto optimality is that, given a set of alternative allocations and a set of individuals, allocation A is an improvement over allocation B only if A can make at least one person better off than B, without making any other worse off.

Based on this, the multi-objective optimisation problem can be defined as to find a vector of decision variables x, which optimises a vector of M objective functions $f_i(x)$ where $i = 1, 2, \ldots, M$. The objective functions are the mathematical description of the optimisation criteria, which are often in conflict with each other.

Without the loss of generality, let us assume that f_i is to be maximised, where $i = 1, 2, \ldots, M$. A decision vector x is said to dominate a decision vector y (also written $x \succ y$) if and only if their objective vectors $f_i(x)$ and $f_i(y)$satisfies:

$$f_i(x) \geq f_i(y) \forall i \in \{1, 2, \ldots, M\}; and$$

$$\exists i \in \{1, 2, \ldots, M\} | f_i(x) > f_i(y)$$

All decision vectors that are not dominated by any other decision vectors are said to form the *Pareto optimal set*, while the corresponding objective vectors are said to form the *Pareto frontier*. Now the multi-objective optimisation problem can be defined

as follows:

Multi Objective Optimisation

Given: a vector of decision variables, x, and a set of objective functions, $f_i(x)$ where $i = 1, 2, \ldots, M$

Problem: maximise $\{f_1(x), f_2(x), \ldots, f_M(x)\}$ by finding the Pareto optimal set over the feasible set of solutions.

Identifying the Pareto frontier is particularly useful in engineering because the decision maker can use the frontier to make a well-informed decision that balances the trade-offs between the objectives.

The multi-objective test case selection problem is to select a Pareto efficient subset of the test suite, based on multiple test criteria. It can be defined as follows:

Multi Objective Test Suite Minimisation

Given: a test suite, T, a vector of M objective functions, f_i, $i = 1, 2, \ldots, M$.

Problem: to find a subset of T, T', such that T' is a Pareto optimal set with respect to the objective functions, f_i, $i = 1, 2, \ldots, M$.

The objective functions are the mathematical descriptions of test criteria concerned. A subset t_1 is said to dominate t_2 when the decision vector for t_1 ($\{f_1(t_1), \ldots, f_M(t_1)\}$) dominates that of t_2. The resulting subset of the test suite, T', has several benefits in regards to the regression testing, as shown in Section 3.3.2.

3.3.2 Properties of 2-Objective Coverage Based Minimisation

Here we instantiate the two objective formulation with code coverage as a measure of test adequacy and execution time as a measure of cost. Thus, code coverage becomes

one of the two objectives, and it should be maximised for a given cost. Time is the other objective, which should be minimised for a given code coverage.

In this instantiation of the problem, should there exist a subset of test suite S with coverage C and execution time T on the Pareto frontier, it means that:

- **T1.** No other subset of S can achieve more coverage than C without spending more time than T.
- **T2.** No other subset of S can finish in less time than T while achieving a coverage that is equal to or greater than C.

This is the implication of Pareto optimality. Rather than obtaining a single answer that approximates the global optimum in the search space for a single objective, we obtain a set of points, each of which denotes one possible way of balancing the two objectives in a globally optimal way. Each member of the Pareto frontier is therefore a candidate solution to the problem, upon which it is not possible to improve.

In the single objective formulation of test suite minimisation, greedy algorithms have been used to maximise coverage. The greedy approach starts with an empty test set as the 'current solution' and iteratively adds a test case which gives the most coverage of those that remain. A variant, additional greedy, improves on this by adding to the current solution the test case that gives the best *additional* coverage to the current solution. Each addition by the greedy algorithm of a new test case to the 'current solution' denotes a candidate element of the Pareto frontier.

Greedy algorithms have proved effective for the single objective formulation, so they make a sensible starting point for the consideration of the multi-objective formulation. In order to optimise both coverage and cost, the additional greedy algorithm will need to be formulated to measure not coverage, but coverage per unit time. This produces a single objective cost cognizant variant of the greedy algorithm, similar to that used by Malishevsky et al. for the single objective prioritisation problem [130].

Suppose that the additional greedy algorithm has chosen a test case t that covers a set of structural elements, s. Let $Cov(s)$ be the coverage of test case t and let $Time(t)$ be the execution time of t. Assume that the selection of t increases the coverage by $\Delta Cov(s)$. By definition, there is no single test case t' (which would cover s') that the algorithm could have chosen, such that $\Delta Cov(s') > \Delta Cov(s)$ and $Time(t') \leq Time(t)$ (otherwise the algorithm would have picked t'). Therefore, the selection of a test case made by the two objective cost cognizant additional greedy algorithm cannot be improved upon by the addition of another single case. However, this leaves open the possibility that there may be a *set* of test cases that, taken together, could have produced a better approximation to the Pareto front.

Let us consider the case of the basic greedy algorithm that selects one test case at a time. It turns out that any selection of a test case made by the additional greedy algorithm can only be improved with respect to **T2**. It is not possible to improve on the selection made by the additional greedy algorithm with respect to **T1**. This observation is stated and proved more formally below.

Proposition 1 (Partial Optimality)**.**
The selection of a test case made by the additional greedy algorithm cannot be improved upon with respect to **T1**.

Proof. Suppose the contrary. That is, let t_1 be a test case that covers a set of structural elements, s_1. Suppose there also exists a pair of test cases, t_2 and t_3, covering s_2 and s_3 respectively, that together improve upon t_1 by achieving more coverage without spending more time. By definition, we have

$$\frac{\Delta Cov(s_2)}{Time(t_2)} < \frac{\Delta Cov(s_1)}{Time(t_1)} \text{ and } \frac{\Delta Cov(s_3)}{Time(t_3)} < \frac{\Delta Cov(s_1)}{Time(t_1)} \tag{3.1}$$

because, otherwise, the additional greedy algorithm would not have selected t_1. From this, it follows that

$$Time(t_1) \cdot (\Delta Cov(s_2) + \Delta Cov(s_3)) < \Delta Cov(s_1) \cdot (Time(t_2) + Time(t_3)) \tag{3.2}$$

However, in order for t_2 and t_3 to be collectively a better choice than t_1 we require t_2 and t_3 to achieve higher increase in coverage, taking no longer than t_1. That is,

$$\Delta Cov(s_2 \cup s_3) > \Delta Cov(s_1) \tag{3.3}$$

and

$$Time(t_2) + Time(t_3) \leq Time(t_1) \tag{3.4}$$

Combining step 3.2 and step 3.4, we get: $\Delta Cov(s_2) + \Delta Cov(s_3) < \Delta Cov(s_1)$. Now, because code coverage is a set theoretic concept, it is not possible for the coverage of the union to be greater than the sum of the coverage of the parts. Therefore we have: $\Delta Cov(s_2 \cup s_3) \leq \Delta Cov(s_2) + \Delta Cov(s_3)$. By transitivity, $\Delta Cov(s_2 \cup s_3) < \Delta Cov(s_1)$, which contradicts step 3.3, so we must conclude that it is not possible to dominate the selection made by the additional greedy algorithm by breaking **T1**. □

	Program Points										Exec. Time
t_1	X	X	X	X	X	X	X	X			4
t_2	X	X		X	X	X	X	X	X	X	5
t_3	X	X	X						X		3
t_4	X	X	X	X	X						3

Table 3.1: An example of a test suite where the additional greedy algorithm produces suboptimal minimisation of test cases

However it is possible to construct an example that shows that the additional greedy algorithm does not produce solutions that are Pareto efficient with respect to **T2**. Such

an example is shown in Table 3.1. The first choice of the additional greedy algorithm will be t_1, which has the additional coverage per unit time value of $\frac{0.8}{4} = 0.2$ (T_2, T_3, T_4 each has 0.18, $0.1\tilde{3}$, and $0.1\tilde{6}$). The second choice will be t_2 with the additional coverage per unit time value of $\frac{0.2}{5} = 0.04$, whereas t_3 and t_4 each has $0.0\tilde{3}$ and 0. At this point, the algorithm achieves 100% coverage in 9 units of time. However, the same amount of coverage is also achievable in 8 units of time by selecting t_2 and t_3, so the subset $\{t_2, t_3\}$ dominates the subset $\{t_1, t_2\}$.

It is indeed possible to extend the greedy approach to consider a pair of test cases, rather than a single test case, at a time, to overcome this problem. This formulation of the greedy approach is often called a *2-way* greedy algorithm. Then, however, it would be possible to construct another counter-exmple that consists of a set of 3 test cases. Eventually, for n test cases, an n-way greedy approach is required to ensure its Pareto-optimality with respect to **T2**. However, the n-way greedy approach would be identical to an exhaustive search, which is not practical.

Furthermore, though the additional greedy algorithm may produce points that are Pareto efficient with respect to **T1**, it does not produce a complete Pareto frontier. The existence of t_4 in the above example demonstrates this. According to the additional greedy algorithm, the first decision point chosen for this example would be the subset of $\{t_1\}$, which achieves 80% coverage in 4 units of time. The subset $\{t_1\}$ is on the Pareto frontier because no other test case can achieve 80% coverage in 4 units of time. However, the subset of $\{t_4\}$ is *also* on the Pareto frontier, because no other test case can achieve 50% coverage in 3 units of time. This point $\{t_4\}$ on the Pareto frontier is ignored by the additional greedy algorithm. As we will see in the next subsection, this issue is important, because it is necessary to produce the most complete approximation to the Pareto front possible in order to exploit the relationship between multi-objective selection and prioritisation.

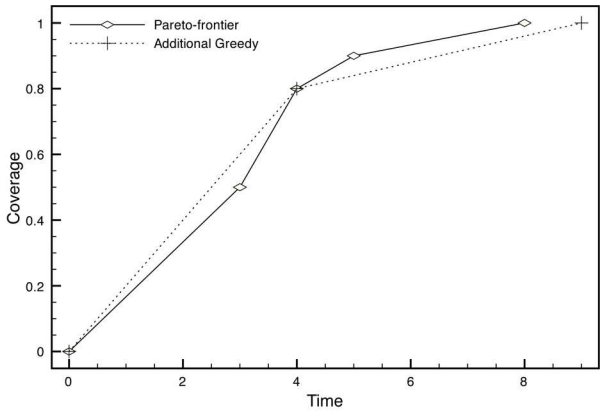

Figure 3.1: Comparison between the Pareto frontier and the results of the additional greedy algorithm from the test data shown in Table 3.1

3.3.3 The Relationship Between Multi Objective Minimisation and Prioritisation

While they are formally different concepts, test suite minimisation and test case prioritisation problems are closely related to each other. Test case prioritisation concerns the most ideal ordering of a given test suite. Since it only changes the order of a given test suite, it is not capable of producing an efficient test case scheduling when the available time is shorter than the total time required by the test suite, assuming that the test suite can be executed in its entirety.

Figure 3.1 shows the result that the additional greedy algorithm produces with the test data shown in Table 3.1, along with the real Pareto frontier of the test data. If the budget allows 9 units of time for the testing, the result of the additional greedy algorithm can be applied with a final coverage of 100%. Now suppose that the budget allows only 6 units of time. From the result of the additional greedy algorithm, the next feasible solution is to execute just T_1, achieving 80% coverage. However, the Pareto frontier tells us that the subset of T_2 can be executed in 5 units of time, achieving 90% coverage. It also shows us that a coverage of 100% is achievable in only 8 units of time. It also

reveals that, should the budget allow only 3 units of time, it is still possible to achieve 50% coverage by executing T_4.

The benefit of knowing the existence of $\{T_2, T_3\}$ and $\{T_4\}$ as candidate selections of test cases becomes clear under the assumption that there is a cost constraint, i.e., testing budget. Prioritisation techniques make no such assumption; they assume that whatever ordering of test cases they produce can be executed in its entirety. However, there can be situations when the exact amount of the available budget is known before the testing begins. Test case prioritisation techniques cannot optimise the testing process in such a situation, because they are not capable of *selecting* test cases. In order to construct an efficient test sequence under cost constraint, an appropriate subset of test cases should be selected first. This subset can *subsequently* be prioritised in order to achieve the ideal ordering among the selected test cases. This way, test case selection and prioritisation techniques can be used in combination in order to achieve more efficient regression testing.

3.4 Empirical Studies

This section explains the experiments conducted to explore the two and three objective formulations of the multi-objective selection problem. Sections 3.4.1 and 3.4.2 set out the research questions and subjects studied. Section 3.4.3 describes the objectives to be optimised. Section 3.4.4 describes the algorithms studied, while Section 3.4.5 explains the mechanisms by which these algorithms will be evaluated in the two empirical studies.

3.4.1 Research Questions

The first three research questions can be answered quantitatively using the approaches described in Section 3.4.5. The last research question is more qualitative in nature.

- **RQ1**: Do the situations theoretically predicted in Section 3.3.2 arise in practice?

That is, does there exist a situation in which the greedy algorithm can be out–performed by solutions that achieve identical coverage in less time? Do there exist situations where the Pareto efficient approaches produce more points on the Pareto front than the greedy algorithm?

- **RQ2**: How well do the greedy and search–based algorithms perform compared to each other and to the global optimum for the 2-objective formulation?
- **RQ3**: How well do the greedy and search–based algorithms perform compared to each other for the 3-objective formulation?
- **RQ4**: What can be said about the shape of the Pareto frontiers, both approximated and optimal? What insights do they reveal concerning the tester's dilemma as to how to balance the trade-offs between objectives?

3.4.2 Subjects

A total of 5 programs were studied: a part of the Siemens suite, including `printtokens`, `printtokens2`, `schedule` and `schedule2`, and the program `space` from the European Space Agency. These programs range from 374 to 6,199 lines of code, and include a real world application. The software artifacts were obtained from Software-artifact Infrastructure Repository (SIR) [41].

Each program has a large number of available test suites. Four test suites were randomly selected for each program; therefore a total of 20 test suites were used as input to the multi-objective Pareto optimisation. The size of the programs and their test suites are shown in Table 3.2.

3.4.3 Objectives

It is not the aim of this thesis to enter into a discussion concerning which objectives are more important for regression testing. We simply note that, irrespective of arguments

Program	Lines of Code	Average Test Suite Size
`printtokens`	726	16
`printtokens2`	570	17
`schedule`	412	8
`schedule2`	374	8
`space`	6199	153

Table 3.2: Test suite size of subject programs studied in Section 3.4

about their suitability, coverage and fault histories are likely candidate objectives for assessing test adequacy and that execution time is one realistic measure of effort.

For the two objective formulation, statement coverage and computational cost of test cases will be used as objectives. The additional objective used in the three objective formulation is the past fault detection history. Each software artifact used in this chapter has several seeded faults (taken from the data available on the SIR [41]), which are associated with the test cases that reveal them. Using this information, it is possible to assign past fault coverage to each test case subset, which corresponds to how many of the known, past faults in the previous version this subset would have revealed.

Physical execution time of test cases is hard to measure accurately. It involves many external parameters that can affect the physical execution time; different hardware, application software and operating system. In particular, any measurement of execution time is likely to be affected by aspects of the environment unconnected to the the choice of test cases. Such factors include concurrent execution, caching and other low-level processor optimisations.

Here we circumvent these issues by using the software profiling tool, Valgrind, which executes the program binary code in an emulated, virtual CPU [147]. The computational cost of each test case was measured by counting the number of virtual instruction codes executed by the emulated environment. Valgrind was created to allow just this sort of precise and unequivocal assessment of computational effort; it allows us to argue that

these counts are directly proportional to the cost of the test case execution.

3.4.4 Algorithms

Two different Pareto efficient genetic algorithms, NSGA-II and its variation were used. NSGA-II is a multi-objective genetic algorithm developed by Deb et al. [38]. The output of NSGA-II is not a single solution, but the final state of the Pareto frontier that the algorithm has constructed. Pareto optimality is used in the process of selecting individuals. This leads to the problem of selecting one individual out of a non-dominated pair. NSGA-II uses the concept of crowding distance to make this decision; crowding distance measures how far away an individual is from the rest of the population. NSGA-II tries to achieve a wider Pareto frontier by selecting individuals that are far from the others. NSGA-II is based on elitism; it performs the non-dominated sorting in each generation in order to preserve the individuals on the current Pareto frontier into the next generation.

A novel variation of NSGA-II, called vNSGA-II, was also developed and implemented. Two major modifications to NSGA-II were made for vNSGA-II. First, the algorithm uses a group of sub-populations that are separate from each other, in order to achieve wider Pareto frontiers. When performing a pairwise tournament selection on individuals that form a non-dominated pair, each of the sub-populations slightly prefers different objectives so that the Pareto frontier can be advanced in all the directions. vNSGA-II also extends the elitism of NSGA-II by keeping a *best-so-far* record of the Pareto frontier separate from the sub-populations.

Two Greedy Algorithms were also implemented. For the two objective formulation, the cost cognizant version of the additional greedy algorithm was implemented. For the three objective formulation, the three objectives were combined into a single objective according to the classical weighted-sum approach. With M different objectives, f_i with $i = 1, 2, \ldots, M$, the weighted-sum approach calculates the single objective, f', as follows:

$$f' = \sum_{i=1}^{M} (w_i \cdot f_i), \sum_{i=1}^{M} w_i = 1$$

Both the additional code coverage per until time and additional past fault coverage per unit time were combined using coefficients of 0.5 and 0.5, thereby giving equal weighting to each objective.

3.4.5 Evaluation Mechanisms

The difficulty of evaluating Pareto frontiers lies in the fact that the absolute frame of reference is the *real* Pareto frontier, which by definition, is impossible to know *a priori*. Instead, a reference Pareto frontier can be constructed and used when comparing different algorithms with respect to the Pareto frontiers they produce. The reference frontier represents the hybrid of all approaches, combining the best of each. It is one of the advantages of Pareto optimality that results for various approaches can be combined in this way.

More formally, let us assume that we have N different Pareto frontiers, P_i with $i = 1, 2, \ldots, N$. A reference Pareto frontier, P_{ref}, can be formulated as follows. Let P' be the union of all P_i with $i = 1, 2, \ldots, N$. Then:

$$P_{ref} \subset P', (\forall p \in P_{ref})(\nexists q \in P')(q \succ p)$$

For the programs from the Siemens suite, the search spaces were sufficiently small to allow us to perform an exhaustive search to locate the true Pareto frontier. This allows us to compare the results from the algorithms to the globally optimal solution in these cases. For the program `space` this was not possible, so the reference Pareto frontier was formed as described.

One of the methods to compare Pareto frontiers is to look at the number of solutions that are not dominated by the reference Pareto frontier. By definition, P_{ref} is not dom-

inated by any of the N different Pareto frontiers, because it consists of the best parts of the different Pareto frontiers. However, each of N different Pareto frontiers may be partly dominated by P_{ref}. Therefore, these N different Pareto frontiers can be compared with each other by counting the number of solutions that are not dominated by P_{ref} in each Pareto frontier.

Another meaningful measurement is the size of each Pareto frontier. Achieving wider Pareto frontiers is one of the important goals of Pareto optimisation. This is particularly of concern in engineering application, because a wider Pareto frontier means a larger number of alternatives available to the decision maker.

Both the number of non-dominated solutions and the size of Pareto frontiers were measured and statistically analysed using Welch's t-test. Welch's t-test is a statistical hypothesis test for two groups with different variance values. It tests the null hypothesis that the means of two normally distributed groups are equal. In the context of this chapter, the null hypothesis is that with two different algorithms, the mean values of the number of solutions that are not dominated by the reference Pareto frontier are equivalent. For these tests the α level was set to 0.95. Significant $p - values$ suggest that the null hypothesis should be rejected in favour of the alternative hypothesis, which states that one of the algorithm produces a larger number of non-dominated solutions.

NSGA-II and vNSGA-II algorithms were both executed 20 times for each test suite to account for their inherent randomness. Both algorithms use single-point crossover and bit-flip mutation. NSGA-II is configured with the recommended setting of $\{population = 100$, and $maximum\ fitness\ evaluation = 25,000\}$ for the Siemens suite. vNSGA-II uses three different sub-population groups of $\{population = 300$, and $maximum\ iteration = 250\}$ for the Siemens suite. For `space`, both algorithm use the setting of $\{population = 1,500$ and $\{maximum\ iteration = 180\}$. In the case of vNSGA-II, this means three sub-populations with 500 individuals.

3.5 Results and Analysis

The results for the 2-objective formulation for the five different subjects are shown in Figure 3.2. The figures are provided for illustration and qualitative evaluation only. For complete quantitative data, see Table 3.3.

In particular, it should be noted that the lines connecting the data points are drawn merely in order to aid the visual comprehension of the plot; no meaning can be ascribed concerning the results that may or may not exist along these lines, apart from the points plotted. In case with vNSGA-II and NSGA-II, a single result was chosen out of the 20 experiments in order to produce readable images. The variance in their complete results over 20 runs can be seen in Table 3.3.

The results from the programs from the Siemens suite confirm the theoretical argument set out in Section 3.3.1; there do exist data points that achieve the same amount of coverage as the additional greedy algorithm, but in less time. The size of Pareto frontiers produced by the Pareto efficient genetic algorithms are larger than those produced by the additional greedy algorithm, giving more information to the tester. In all four smaller programs, the Pareto efficient genetic algorithms produce subsets of test cases that can be executed in fewer than 200 units of cost, something for which the additional greedy algorithm is incapable. These results provide a positive answer to **RQ1**.

It can also be observed that NSGA-II is capable of identifying the entire reference frontier, producing the exhaustive result. The results from vNSGA-II are not always exhaustive, but they still outperform the additional greedy algorithm.

However, the result for the (larger) program `space` shows the contrary; the additional greedy algorithm performs very well, dominating the rest of the algorithms. NSGA-II, which is very competitive with the smaller programs, manages to produce results that are close to those produced by the additional greedy algorithm but none of the points on its approximation to the Pareto frontier dominates those found by the additional

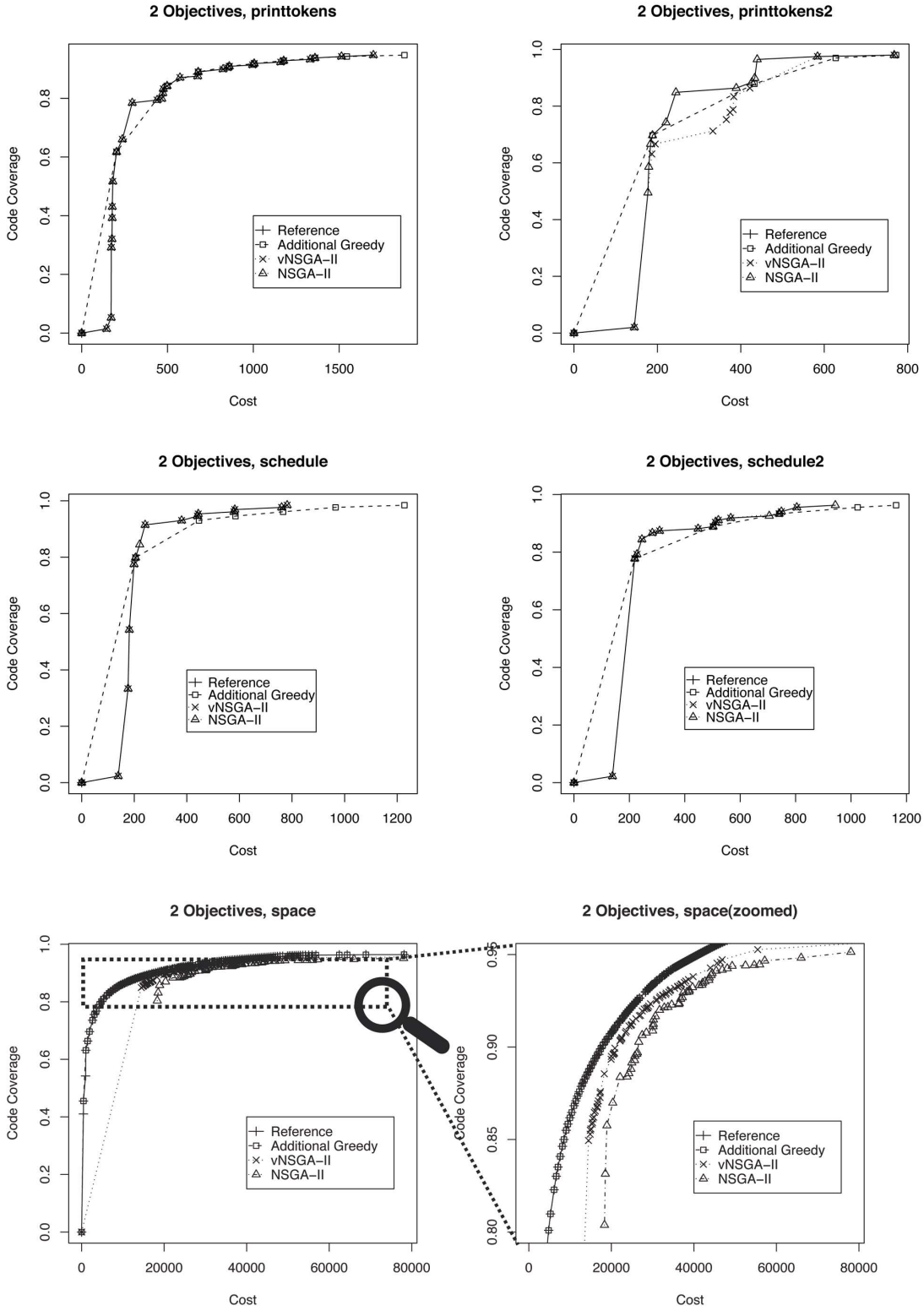

Figure 3.2: Plot of the Pareto frontier for the two objective formulation. With the Siemens suite, the results from the additional greedy algorithm are dominated by the reference Pareto frontier obtained by an exhaustive search, which NSGA-II is also capable of finding. However, in the zoomed plot for the program `space`, it can be observed that the additional greedy algorithm dominates the rest of the algorithms.

greedy algorithm. vNSGA-II partly dominates NSGA-II, but its results are still inferior to those of the additional greedy algorithm. The good performance of the additional greedy algorithm suggests that the existing test case prioritisation techniques are capable of producing solutions that are strongly Pareto efficient. This is a very attractive finding, given the computations efficiency of the greedy algorithms, compared to the alternatives.

These findings provide a mixed message for the answer to **RQ2**. The data show that the additional greedy algorithm may be dominated by the Pareto efficient genetic algorithms, but also that, for some programs the additional greedy algorithm produces the best results. This suggests that for optimal quality test data selection it may be advisable to combine the results from greedy and evolutionary algorithms. This is one of the attractive aspects of the Pareto efficient approach; results from several algorithms can be merged to form a single Pareto front that combines the best of all approaches.

Figure 3.3 shows the results for the three objective formulation. The 3D plots display the solutions produced by the weighted-sum additional greedy algorithm (depicted by a line in the figures), and the reference Pareto frontier (depicted by square–shaped points). The weighted-sum additional greedy algorithm produces very strong results because the line can be seen to connect the data points forming the reference Pareto frontier, meaning that the solutions from the weighted-sum and additional greedy algorithm form a part of the reference Pareto frontier (which is later confirmed by a statistical analysis).

These results suggest that the answer to **RQ3** is also mixed. Even where there are more than 2 objectives, the greedy approach is capable of reasonable performance. Therefore, a combination of results may be appropriate. Of course, these findings will depend upon the three objectives chosen. More work is required to experiment with other objectives. It is not possible to extrapolate from these results to conclude than the additional greedy algorithm will perform well for any 3 objective instantiation of the multi-objective test case selection problem. For example, the strong results that we have been able to obtain may be a result of the relative sparseness of the fault history data,

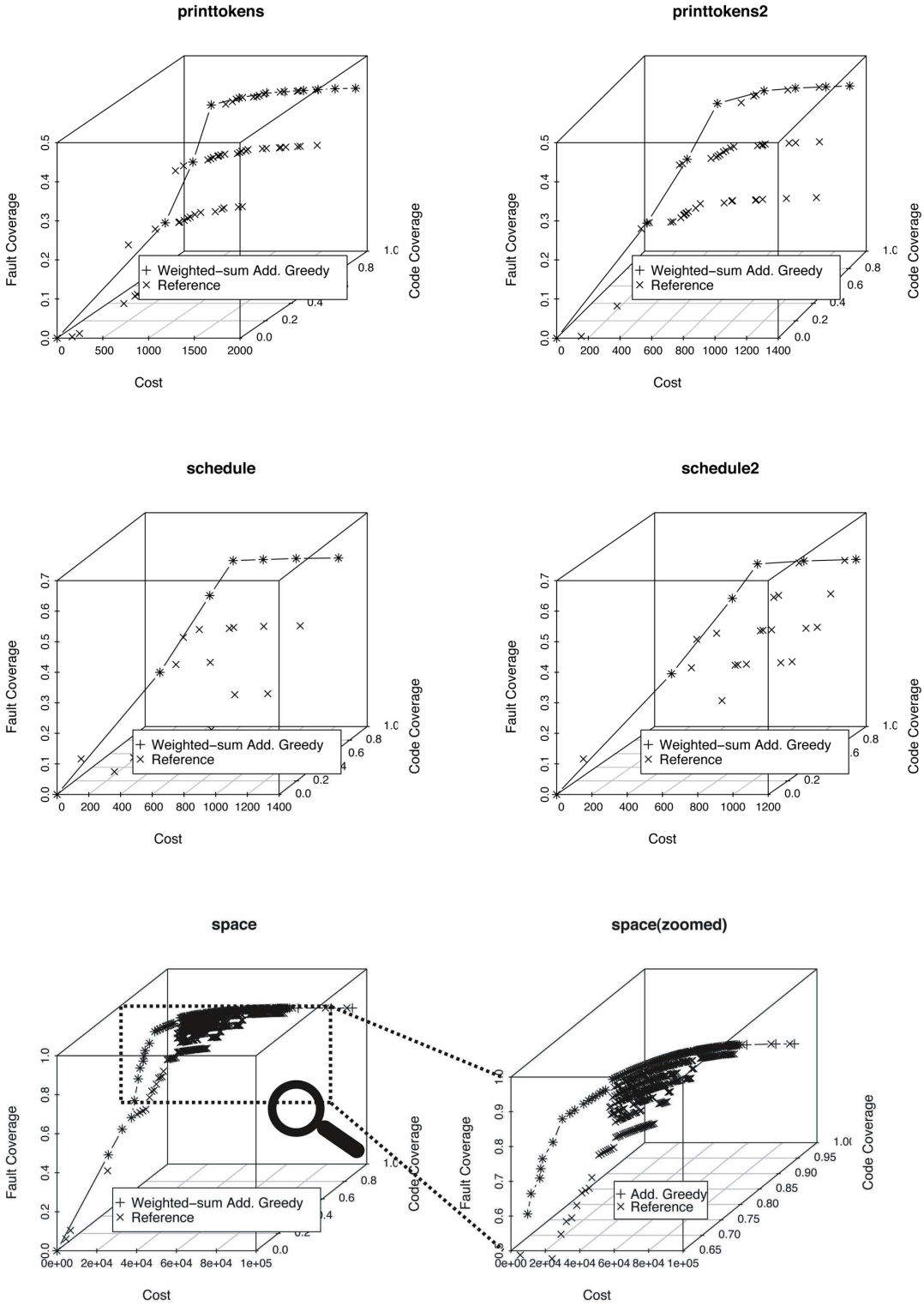

Figure 3.3: Plot of the Pareto frontier for the three objective formulation. The solid line connects the results from weighted-sum additional greedy, while all the shown points correspond to the reference Pareto frontier, which is a surface in 3d.

which may favour the additional greedy approach.

In order to provide a more concrete quantitative analysis of the answers to **RQ2** and **RQ3**, we compare the results obtained using tests for statistical significance. For the Siemens suite programs, NSGA-II shows the best performance by producing the entire reference Pareto frontier. The results from NSGA-II for the Siemens suite are also very stable; the variance in the size of the Pareto frontier produced by NSGA-II is 0. On the other hand, the results from vNSGA-II show some variance, and therefore are compared to the additional greedy algorithm using the t-test analysis with the confidence level of 95% and using the null hypothesis that there is no difference in the results for $\bar{n}_{vNSGA-II}$ and $\bar{n}_{AdditionalGreedy}$ and the alternative hypothesis that $\bar{n}_{vNSGA-II}$ is greater than $\bar{n}_{AdditionalGreedy}$. The observed $p - values$ for these t–tests are significant at the 95% level, confirming the alternative hypothesis.

However, the weighted-sum additional greedy algorithm produces the best result with `space`. For the analysis of vNSGA-II results from `space`, the alternative hypothesis is that $\bar{n}_{vNSGA-II}$ is smaller than $\bar{n}_{AdditionalGreedy}$. The observed $p-values$ are significant at the 95% level, confirming the alternative hypothesis. For some test suites of `space`, the results were constant, making the t-test inapplicable. For the program `space`, the Pareto frontiers produced by NSGA-II are completely dominated by the results from the additional greedy algorithm.

Both of the genetic algorithms produce much wider Pareto frontiers than the additional greedy algorithm, which is expected because they are designed to produce Pareto frontiers. However, in terms of the number of solutions that are not dominated by the reference frontier, statistical analysis confirms the results shown in Figure 3.2 and 3.3. With smaller programs, NSGA-II performs significantly better than the others in both two and three objective formulations, while `space` shows the contrary. However, the higher deviation observed in the results of vNSGA-II with test suite T_2 of `space` suggest that both the random nature of the genetic algorithm and the composition of a par-

ticular test suite may affect the result; further research on wider range of subjects will confirm or refute this.

Turning to the last research question, **RQ4**, a more qualitative analysis is required. This is made possible by the visualisations of the solutions plotted in Figures 3.2 and 3.3. It can be seen that the shapes of the lines and the reference Pareto frontiers are relatively similar to each other across all programs, suggesting a similar relationship between coverage and fault detection for these programs. The shape is an interesting observation on the relation between the code coverage and past fault coverage, because it seems to illustrate a relatively strong correlation between the two objectives. Such a correlation may suggest that the concerned faults are not concentrated in a limited part of the code. There also appear to exist a point on the line in every program, where the rate of increase in the fault coverage changes. Such *elbow points* are considered important in the study of Pareto optimality. They indicate points of particular interest where the balance of trade-offs inherent between the objectives.

In the case of test suite minimisation, the location of this elbow point may tell us the percentage of faults that require certain amount of code coverage to be detected. The selections *below* the elbow point generally contain smaller numbers of test cases, which results in the limited fault detection capability; once a sufficient amount of cost is available, combinations of test cases can be picked up, which improves the rate of the past fault coverage. These results provide evidence to suggest a 'critical mass' phenomenon in test case selection. These observations form a partial answer to **RQ4**, but more data is required to see whether this critical mass phenomenon is generic to test case selection, or whether it is merely an artefact of the set of programs that have been chosen to be studied.

2 Objectives											
Program	Suite	vNSGA-II				NSGA-II			Additional Greedy		
		$\bar{n}$	σ	p	Avg. Size	$\bar{n}$	σ	Avg. Size	$\bar{n}$	σ	Size
printtokens	T_1	13.00	2.51	3.5e-16	16.75	23.00	0.00	23.00	2.00	0.00	10.00
	T_2	14.15	2.08	6.0e-13	21.30	33.00	0.00	33.00	8.00	0.00	11.00
	T_3	13.70	3.61	4.3e-08	18.20	30.00	0.00	30.00	8.00	0.00	10.00
	T_4	10.15	2.16	2.4e-14	13.40	19.00	0.00	19.00	2.00	0.00	8.00
printtokens2	T_1	9.05	2.85	9.9e-18	12.55	24.00	0.00	24.00	4.00	0.00	6.00
	T_2	14.00	1.97	1.0e-12	16.00	25.00	0.00	25.00	7.00	0.00	7.00
	T_3	7.75	1.52	3.1e-13	9.40	14.00	0.00	14.00	2.00	0.00	5.00
	T_4	8.60	1.96	1.2e-09	11.90	24.00	0.00	24.00	4.00	0.00	7.00
schedule	T_1	8.90	1.59	2.6e-14	9.00	11.00	0.00	11.00	2.00	0.00	6.00
	T_2	11.95	1.47	$<$ 2.2e-16	12.00	15.00	0.00	15.00	2.00	0.00	7.00
	T_3	10.30	1.13	6.5e-15	10.30	12.00	0.00	12.00	5.00	0.00	5.00
	T_4	10.45	1.47	$<$ 2.2e-16	10.70	13.00	0.00	13.00	2.00	0.00	6.00
schedule2	T_1	7.75	0.91	$<$ 2.2e-16	7.75	9.00	0.00	9.00	2.00	0.00	5.00
	T_2	12.70	1.49	$<$ 2.2e-16	13.15	17.00	0.00	17.00	4.00	0.00	6.00
	T_3	8.70	1.38	3.6e-15	8.70	11.00	0.00	11.00	2.00	0.00	6.00
	T_4	8.40	1.19	5.2e-16	8.45	10.00	0.00	10.00	2.00	0.00	6.00
space	T_1	1.40	1.56	$<$ 2.2e-16	99.50	0.00	0.00	55.55	117.00	0.00	117.00
	T_2	1.05	0.22	$<$ 2.2e-16	107.55	0.00	0.00	54.15	118.00	0.00	118.00
	T_3	1.00	0.00	N/A	94.90	0.00	0.00	55.55	92.00	0.00	119.00
	T_4	1.00	0.00	N/A	104.45	0.00	0.00	54.25	120.00	0.00	121.00

Table 3.3: Average number of solutions that are not dominated by the reference Pareto frontier($\bar{n}$), standard deviation(σ), and the size of Pareto frontier for the two objective formulation

3 Objectives											
Program	Suite	vNSGA-II				NSGA-II			Weighted-sum Greedy		
		$\bar{n}$	σ	p	Avg. Size	$\bar{n}$	σ	Avg. Size	$\bar{n}$	σ	Size
printtokens	T_1	11.95	3.02	1.6e-04	15.20	23.00	0.00	23.00	9.00	0.00	9.00
	T_2	25.25	4.78	2.5e-12	39.60	65.65	1.42	66.70	9.00	0.00	11.00
	T_3	20.80	3.30	9.0e-13	26.70	51.00	0.00	51.00	9.00	0.00	9.00
	T_4	13.55	2.78	6.0e-14	21.25	33.00	0.00	33.00	2.00	0.00	8.00
printtokens2	T_1	21.75	4.84	1.6e-12	27.70	55.00	0.00	55.00	5.00	0.00	7.00
	T_2	28.40	5.37	3.1e-13	35.20	59.75	0.64	60.60	8.00	0.00	8.00
	T_3	14.40	2.54	3.6e-12	16.15	24.00	0.00	24.00	6.00	0.00	6.00
	T_4	17.45	3.50	1.2e-10	25.40	48.00	0.00	48.00	8.00	0.00	8.00
schedule	T_1	12.05	1.82	3.2e-12	12.50	16.00	0.00	16.00	6.00	0.00	6.00
	T_2	16.90	2.25	2.1e-14	18.20	23.00	0.00	23.00	7.00	0.00	7.00
	T_3	12.80	1.32	1.2e-15	13.20	16.00	0.00	16.00	6.00	0.00	6.00
	T_4	14.20	1.32	$<$ 2.2e-16	14.45	18.00	0.00	18.00	6.00	0.00	6.00
schedule2	T_1	11.20	0.95	$<$ 2.2e-16	11.40	14.00	0.00	14.00	5.00	0.00	5.00
	T_2	21.30	2.18	$<$ 2.2e-16	22.30	29.00	0.00	29.00	6.00	0.00	6.00
	T_3	12.00	1.62	3.0e-14	12.45	16.00	0.00	16.00	5.00	0.00	5.00
	T_4	11.50	1.19	4.2e-16	11.60	14.00	0.00	14.00	5.00	0.00	6.00
space	T_1	1.50	1.82	$<$ 2.2e-16	268.65	0.00	0.00	119.20	118.00	0.00	118.00
	T_2	19.05	23.50	2.3e-14	263.30	0.00	0.00	119.95	119.00	0.00	122.00
	T_3	1.00	0.00	N/A	208.55	0.00	0.00	96.10	114.00	0.00	119.00
	T_4	2.80	3.65	$<$ 2.2e-16	183.15	0.00	0.00	88.80	67.00	0.00	122.00

Table 3.4: Average number of solutions that are not dominated by the reference Pareto frontier($\bar{n}$), standard deviation(σ), and the size of Pareto frontier for the three objective formulation

3.5.1 Threats to Validity

Threats to internal validity concern the factors that might have affected the multi-objective optimisation techniques used in the study. One potential concern involves the accuracy of the instrumentation of the subject software, e.g. the correctness of the coverage information. To address this, a professional and commercial software tool (Cantata++ from IPL ltd. [1]) was used to collect code coverage information. The fault coverage information was extracted from SIR - a well-managed software archive [41]. Precisely determined computational cost was used in place of the physical execution time in order to raise the precision of the cost information using the Valgrind profiler [147].

Another potential internal threat comes from the selection and optimisation of the meta-heuristic techniques themselves. No particular algorithm is known to be effective for the multi-objective test case selection problem. However the genetic algorithm used in this study is known to be effective for a wide range of multi-objective problems [33, 37], and can serve as a basis for the future research.

Welch's t-test, which is used in the empirical study, assumes a parametric distribution of the samples. While the exact distribution of the sample (the number of points on the Pareto frontier) is not known, it will approximate a normal distribution with a sufficiently large sample size according to the central limit theorem [161].

Threats to external validity concern the conditions that limit generalisation from the result. The primary concern for this study is the representativeness of the subjects that were studied. This threat can be addressed only by additional research using a wider range of software artifacts and optimisation techniques.

3.6 Conclusions

This chapter introduced the concept of Pareto efficient multi-objective optimisation to the problem of test suite minimisation. It described the benefits of Pareto efficient multi-

objective optimisation, and presented an empirical study that investigated the relative effectiveness of three algorithms for Pareto efficient multi-objective test suite minimisation. The empirical results obtained reveal that greedy algorithms (which perform well for single objective formulations) are not always Pareto efficient in the multi-objective paradigm, motivating the study of meta-heuristic search techniques.

Chapter 4

Test Suite Latency

4.1 Introduction

Test suite minimisation aims to reduce the number of test cases that need to be applied, while retaining identical or near-identical satisfaction of a chosen test adequacy criterion. Test suite minimisation is increasingly important, because of the tendency for test suite size to grow over time [70]. This growth in test suite size comes from a variety of sources, including the use of capture replay tools, improvements in test data generation techniques and procedures for capturing customer–created test cases arising from their use of deployed software. Test suite minimisation provides a mechanism for managing the size of the test suite when there are insufficient resources available to apply all available test cases during a period of testing activity.

For example, a large IBM middleware product has a test suite of 20,000 test cases which take 10 days to run [227]. IBM has an automatic regression test selection mechanism that chooses, for each regression test, a subset of the test cases to run. The algorithm first determines the relevant tests given the code that was changed. However, it then needs to further minimize within these selected test cases in order to achieve a test set that can be executed within the time available (which is normally eight to twelve

hours).

IBM's experience with repeated application of test suite minimisation for multiple iterations revealed a problem: minimised test suites have a tendency to 'wear out'. That is, some of the test cases execute parts of the code that have not changed (as well as parts that have). However, for deterministic software systems, repeated execution of unchanged sections of code with the same input cannot, by definition, reveal any new faults provided that the rest of the environment do not change. As the number of previously used test cases increases, the test suite 'wears out' its ability to reveal additional faults. The more a test suite contains test cases that have already been executed on previous versions of the system, the less the test suite is likely to reveal.

In order to avoid wear out, minimisation strategies can be adapted to select different subsets of the test pool to form a new test suite to be applied at each regression cycle. It is trivial to adapt test minimisation algorithms to avoid re-selection of previously used test cases, thereby yielding a novel minimised test suite on each iteration. However, at each iteration, the algorithm may achieve lower coverage because it cannot re-use test cases that have already been executed. Some parts of the system may have very few available test cases that are able to cover them. This raises an important question: How many times can a test pool allow repeated reduction that results in a novel set of test cases?

This thesis introduces the term *latency* to capture this property. The more a test suite allows for selection of novel minimised test suites, the higher is its latency. The use of word 'latency' is overloaded in computer science, imbuing it with possible associations that are unwanted in the context of this work. The dictionary definition of the word is the one intended in this thesis:

Latent('leitənt) *a.* Hidden, concealed; present or existing, but not manifest, exhibited, or developed [3].

The rest of the chapter[1] is organised as follows. Section 4.2 presents formal definitions and theoretical results that underpin the subsequent empirical studies. Section 4.3 presents an empirical study of test suite latency for open source programs. Section 4.4 introduces the strategy and algorithms for combining test suite reduction and generation, which is evaluated in Section 4.5. Section 4.6 discusses the potential threats to validity, and Section 4.7 concludes.

4.2 Problem Statement

Test suite minimisation (sometimes *reduction*) is the problem of selecting a subset of a given test suite in order to reduce the effort required to execute the test cases [171]. Throughout this chapter, the remaining unselected test cases are referred to as the 'retained' set, since they may be retained for subsequent selections. Some test suite quality metrics, for example structural code coverage, monotonically decrease as the number of retained test cases decreases. A test suite minimisation technique is said to be monotonically decreasing with respect to a quality metric, if the repeated application of the minimisation technique results in monotonically decreasing values of the quality metric. The latency of a test suite is defined as the maximum number of times a minimisation technique can be applied to the retained portion of the test suite before the quality metric falls below a predefined threshold level. The definitions below formalise these concepts, facilitating theoretical study of monotonicity and latency.

Let $\mathcal{S}$ be the initial test suite available for testing. Let S be the set of all subsets of $\mathcal{S}$.

Definition 4. ***Test Suite Minimisation Procedure***

[1]This chapter is an extended version of the author's SBST paper: S. Yoo, M. Harman and S. Ur, Measuring and Improving Latency to Avoid Test Suite Wear Out, *Proceedings of the 2009 International Workshop on Search-Based Software Testing*, Denver, CO, USA, pages 101-110, 2009. Best Paper Award.

A test suite minimisation procedure τ *is a relation in* $S \leftrightarrow (S \times S)$ *such that* $s\ \tau\ (a, r) \Rightarrow a \cup r = s$*. If additionally* $\forall s.\ s\ \tau\ (a, r) \Rightarrow a \cap r = \emptyset$*, then* τ *is said to be non-overlapping.*

Thus, a minimisation procedure τ is a relation, where $s\ \tau\ (a, r)$ means that when procedure τ is applied to a test suite s, the subset a of s is applied to the program under test, while the set r is retained for subsequent regression testing. If $a \cap r \neq \emptyset$ then some of the selected test cases will be reused in subsequent selections.

Definition 5. ***Applied Test Cases***

App is a function in $(S \leftrightarrow S \times S) \rightarrow (S \rightarrow S)$ *such that* $s\ App(\tau)\ a \Leftrightarrow s\ \tau\ (a, r)$

Definition 6. ***Retained Test Cases***

Ret is a function in $(S \leftrightarrow S \times S) \rightarrow (S \rightarrow S)$ *such that* $s\ Ret(\tau)\ r \Leftrightarrow s\ \tau\ (a, r)$

$App(\tau)$ is the projection of a procedure τ onto the used portion of the test suite; $Ret(\tau)$ is the projection of a procedure τ onto the retained portion of the test suite.

Definition 7. ***Repeated Minimisation***

Repeated minimisation from s *using* τ *over* n *times* $(n \geq 1)$ *is denoted by* τ^n *and defined as follows:*

$$\tau^1 = \tau$$

$$s\ \tau^{n+1}\ (a, r)\ \textit{iff}\ s\ \tau^n\ (a', r')\ \textit{and}\ r'\ \tau\ (a, r)$$

Repeated minimisation is an iterative procedure of applying a minimisation technique τ to the retained portion of the test suite from the previous minimisation.

Definition 8. ***Quality Metric***

A quality metric μ *is a function in* $S \rightarrow \mathbb{R}$ *that measures the quality of a set of test cases. If additionally* $\forall s' \subseteq s.\ \mu(s') \leq \mu(s)$*, then* μ *is said to be monotonically decreasing.*

Definition 9. ***μ-Optimality***

A test suite minimisation τ is μ-optimal if and only if $\forall s.\ \mu(App(\tau(s))) = max\{\mu(s') | s' \subseteq s\}$.

Proposition 2. *If μ is monotonically decreasing then a non-overlapping μ-optimal τ is guaranteed to be monotonically decreasing with respect to μ, that is,*

$$\mu(App(\tau^{n+1}(s))) \leq \mu(App(\tau^n(s))).$$

proof Let a and r be sets of test cases such that $s\ \tau^n\ (a, r)$. By definition of τ, $App(\tau^{n+1}(s)) = App(\tau(r))$ and $App(\tau^n(s)) = App(\tau(a \cup r))$. By definition of monotonically decreasing μ, $r \subseteq a \cup r \Rightarrow \mu(r) \leq \mu(a \cup r)$. By definition of μ-optimality, $\mu(App(\tau(r))) = \mu(r)$ and $\mu(App(\tau(a \cup r))) = \mu(a \cup r)$. Therefore $\mu(App(\tau^{n+1}(s))) \leq \mu(App(\tau^n(s)))$.

For a test suite minimisation procedure τ that is monotonically decreasing with respect to a quality metric μ, the latency of a given test suite, s, against τ is defined as follows.

Definition 10. ***Latency***

A latency measure, λ, is a function in

$$(S \to \mathbb{R}) \to \mathbb{R} \to (S \leftrightarrow (S \times S)) \to S \to \mathbb{N}$$

$$\lambda\mu\alpha\tau s = \textit{largest } n \textit{ such that } \mu(Ret(\tau)^n s) \geq \alpha$$

That is, the latency $\lambda\mu\alpha\tau s$ of a test suite s with respect to a quality metric μ and threshold quality for adequacy α and a test suite minimisation procedure τ is the largest number of times the procedure τ can be repeatedly applied to the retained portion of s before the quality (as measured by μ) falls below α. A test suite with a higher λ value is capable of providing multiple disjoint subsets of test cases with a quality metric value above the threshold, which makes the test suite more latent. If a test suite satisfies its given testing requirements (such as full branch coverage), then its measured latency

should be at least 1. However, it would be advantageous to achieve λ value of 2 or higher so that the tester can have multiple test case subsets that achieve the same testing requirements.

4.3 Monotonicity & Overlap Study

The research questions for the monotonicity and overlap empirical study are: **RQ1. Monotonicity Measurement:** Are there test suites with poor latency? How does the monotonically decreasing property of quality metrics manifest itself in real world programs and their test suites? **RQ2. Overlap Effect:** How much increase in latency can be achieved by loosening the non-overlapping constraint so that some test cases are allowed to be reused across consecutive test suite minimisations?

The ideal criterion for test suite minimisation is the highest possible fault detection capability. However, the fault detection information is not available before the testing finishes. As a result, structural code coverage is often used as a readily available surrogate. Each test case is said to *cover* different parts of the program, e.g., statements, branches or blocks, that it executes.

Proposition 3. *Structural code coverage is a monotonically decreasing quality metric. The proof is trivial in this case: it is not possible to increase the code coverage achieved by a test suite by removing a test case.*

From Section 4.2, coverage is a monotonically decreasing quality metric. This means that any non-overlapping coverage-optimal test suite minimisation technique also yields a monotonically decreasing coverage quality metric.

This study uses the widely studied additional greedy algorithm as the test suite minimisation technique [53, 54, 130, 178]. Maximising coverage achieved by testing is a set cover problem. The goal of test suite minimisation is to have the smallest set of test cases that covers the entire program. Greedy algorithms are known to be efficient,

producing solutions to set cover problems of size n that are within $\ln n$ of the global optimum [97].

Proposition 4. *The additional greedy algorithm is a non-overlapping μ-optimal test suite minimisation procedure when $\mu(s)$ is the coverage achieved by s.*

proof: The non-overlapping property of the additional greedy algorithm is obvious by the definition of the algorithm. For the μ-optimality, assume the contrary, that is, $\exists s' \subset s.\mu(s') > \mu(App(\tau(s)))$. Since code coverage is monotonically decreasing, $s' \subseteq s \Rightarrow \mu(s') \leq \mu(s)$. Therefore $\mu(App(\tau(s))) < \mu(s') \leq \mu(s)$, which means that there exists a test case, t, such that $t \in s, t \notin App(\tau(s)), \mu(App(\tau(s))) < \mu(\{t\} \cup App(\tau(s)))$. However, such t cannot exist if τ is the additional greedy algorithm; otherwise the algorithm would have chosen t after choosing the subset $App(\tau(s))$. Therefore, the additional greedy algorithm must be μ-optimal when μ measures code coverage.

Note that other non-greedy minimisation techniques may not be monotonically decreasing. While repeated application of any minimisation approach to a finite test suite must ultimately result in zero coverage, this does not mean that coverage will necessarily decrease monotonically.

4.3.1 Experimental Design

Six program test suites have been analysed for their level of latency. The programs were retrieved from Software-artifact Infrastructure Repository(SIR) along with test suites for each program [41]. The size of programs and test suites are shown in Table 4.1.

The programs are analysed for levels of latency against the greedy test suite minimisation. The quality metric μ of the selected subsets is the statement coverage that each selected subset achieves, since this is one of the weakest coverage criteria; if even statement coverage latency cannot be achieved then clearly there is a need for latency improvement. Coverage information was obtained using `gcov` profiler tool from `gcc`.

Program	Lines of code	Test suite size
`printtokens`	726	17
`flex`	15,297	567
`grep`	15,633	806
`gzip`	8,889	213
`sed`	19,737	370
`space`	6,199	156

Table 4.1: Test suite sizes of subject programs studied in Section 4.3

It should be noted that the test suites analysed were not necessarily created with repeated selection of test cases in mind. Therefore the observations that the coverage drops quickly is not a reflection on the overall quality of the test suites studied. However, the results do illustrate the problem that can be raised when repeated selection is not considered at test suite generation time.

One potential method addressing coverage degradation is to allow a certain level of test case overlap between minimisations. This allows consecutive minimisations to share test cases that may contribute to high quality metric value of the selected subset. Test suites which can retain coverage potency by allowing a degree of overlap can only be said to be 'weakly' latent, because some test case re-use is allowed, as determined by the overlap level. To explore the degree to which overlap allows increased coverage, four different levels of overlap are analysed, each allowing $\{10\%, 30\%, 50\%, 70\%\}$ reuse between test suite minimisations. The reusable test cases are determined randomly. The latency analysis is performed 30 times for each test suite with average results presented.

4.3.2 Results and Analysis

Figure 4.1 shows the result of the latency analysis. As the additional greedy algorithm is repeatedly applied to the test suite, the maximum coverage achieved monotonically decreases, providing an answer to **RQ1**. The rate of decrease is generally higher in the region of early iterations, showing that each test suite contains a small number of test

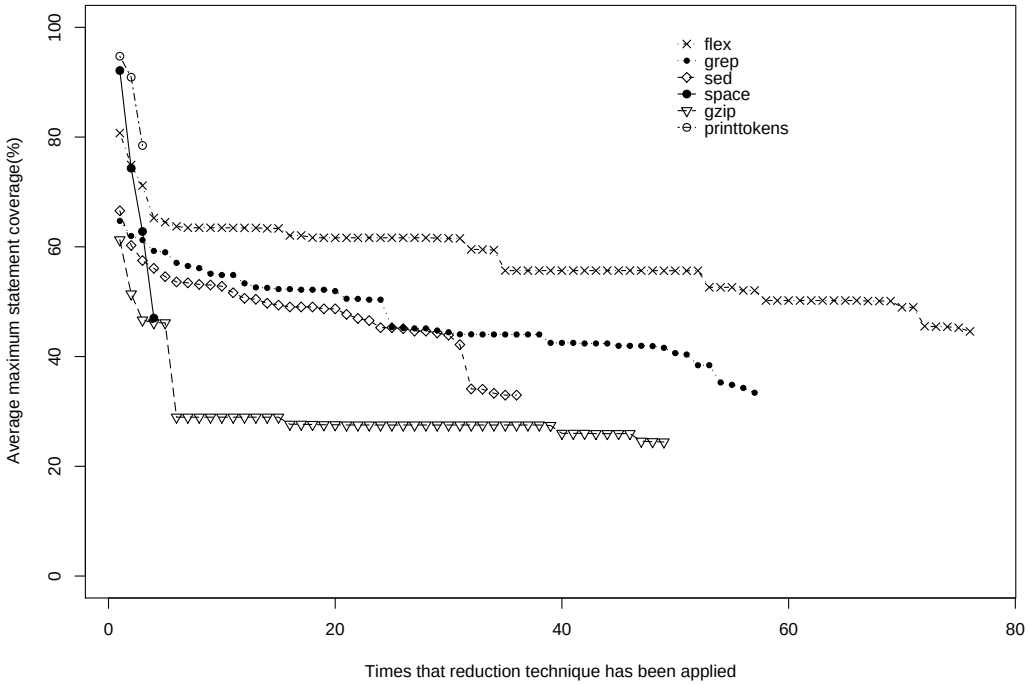

Figure 4.1: Latency Analysis with No Overlap : The greedy test suite minimisation technique is repeatedly applied to test suites, with no overlap between repeated minimisations. Notice that only 4 minimisations are possible for the program `space` and that coverage drops dramatically for all test suites considered.

cases that cover the *hard-to-reach* regions of the programs.

Two programs in particular, `printtokens` and `space`, show an interesting contrast to other programs. The very first iteration on `printtokens` and `space` achieves over 94% and 92% of coverage separately, which is higher than that achieved for any of the other programs. This means that, as long as the tester is concerned with obtaining only a *single* set of test cases to execute, `printtokens` and `space` have very satisfactory test suites. However, the coverage of both programs deteriorates very quickly with repeated minimisations. Should the tester want more than a single 'once and for all' test set, then each consecutive minimisation from the test suite will necessarily lead to the re-execution of a great many test cases. This shows that even a test suite that achieves high level of test adequacy can be vulnerable to low latency.

Note that coverage also drops to a low level for `flex` and `grep`. However, repeated achievement of this low level coverage is possible with novel test cases at each selection. In terms of novelty of testing, `flex` also happens to have the test suite that yields the largest number of different test case subsets.

Table 4.2 shows the latency level of each program with threshold α separately set to 0.05 and 0.1 below that of the first selection, which is denoted by q. The test suite for `grep` shows the highest latency, while the test suites for `space` and `gzip` do not yield more than a single test case subset above the given threshold values.

Program	$\lambda, \alpha = q - 0.05$	$\lambda, \alpha = q - 0.1$
`printtokens`	2	2
`flex`	1	3
`grep`	3	11
`gzip`	1	1
`sed`	1	3
`space`	1	1

Table 4.2: Degree of latency with $\alpha = q - \{0.05, 0.1\}$

Figure 4.1 showed that coverage drops dramatically as repeated minimisations are

made. In this figure no overlap was permitted, ensuring that each new minimisation enjoys an entirely fresh set of test cases. This requirement for 100% 'freshness' could be the cause of the dramatic drop in coverage, suggesting that a more relaxed approach, allowing *some* overlap, may improve coverage. This is the motivation for **RQ2** which concerns the effects (on coverage achievable) of various levels of overlap allowance.

Figure 4.2 shows how the latency of test suites changes when different percentages of the selected test cases are allowed to be reused after each application of the minimisation technique. The results plotted in Figure 4.2 are average values of coverage achieved by each of 30 executions. The allowance of overlap affects the test suites of `printtokens` and `space` positively, resulting in slower degradation of coverage. However, it is interesting to observe that even allowing 70% of the selected test cases to overlap between minimisations does not affect the latency of test suites for other programs in any noticeable way. Overall, the results from the overlapping formulation of the analysis show that coverage decreases dramatically even when overlaps are allowed, which provides an answer to **RQ2**.

4.4 Latency Enhancement Strategy

The results in Section 4.3.2 show that even allowing high levels of overlapping does not improve test suite latency noticeably. This motivates the consideration of ways of enhancing latency. In order to propose a simple and efficient method of enhancing low latency of test suites, the present study raises the following question: if there are known test cases that are effective at achieving coverage, are there different but similar test cases that achieve the same coverage? More formally, given a set of test cases, a, such that $s\ \tau\ (a, r)$ and $\mu(a)$ satisfies the testing criteria, is it realistic to assume that there may exist a' such that a' is similar to a and $\mu(a') = \mu(a)$?

If so, then the enhancement of low latency of a test suite can benefit from knowledge

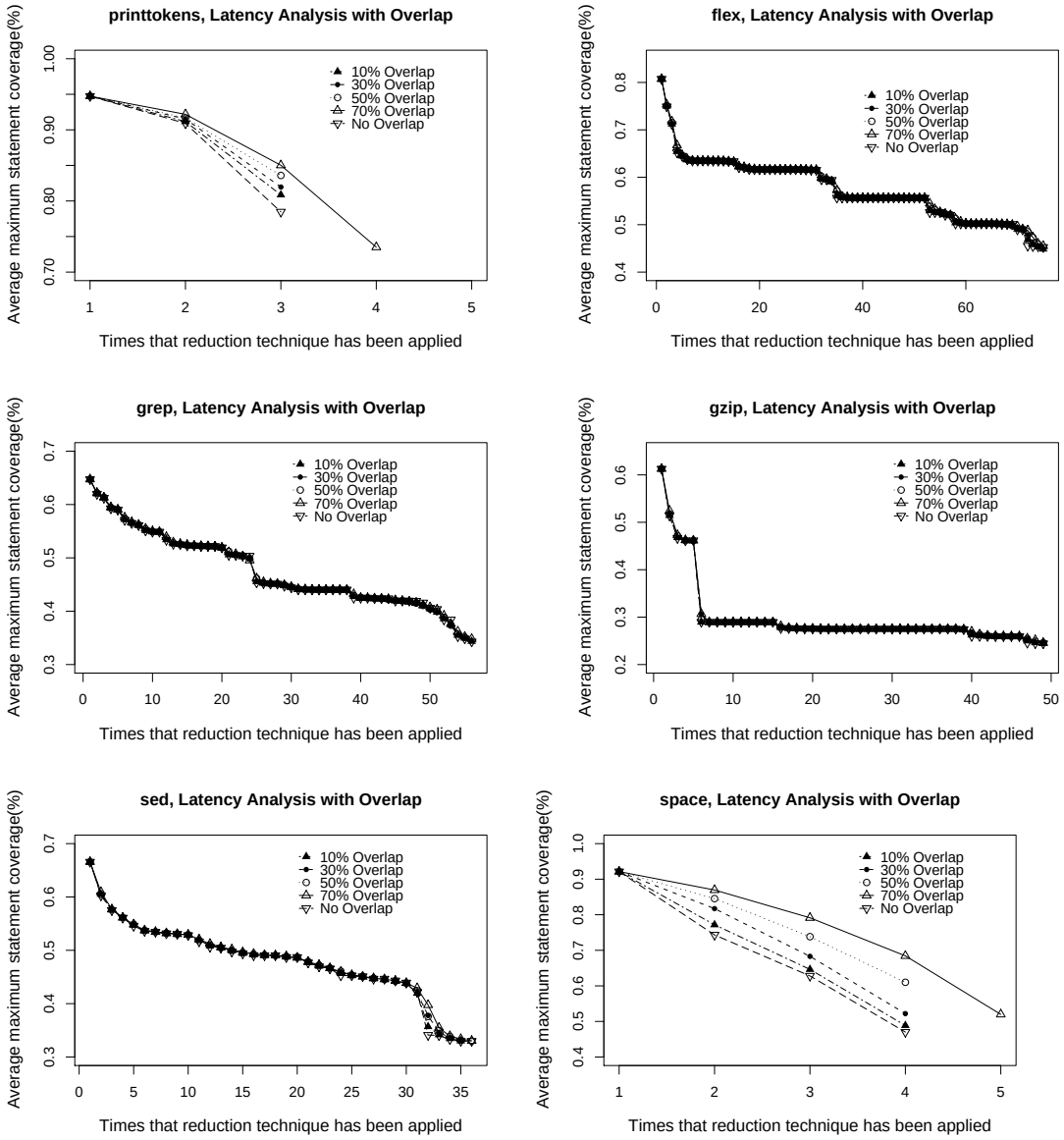

Figure 4.2: Weak Latency Analysis (With Overlapping Test Subsets Allowed) : As more test cases are re-used by allowing a higher level of overlapping, test suites of `printtokens` and `space` show some achievement of weak latency. However, even allowing overlapping does not noticeably affect other programs. This can be seen because coverage profiles are almost identical regardless of overlap allowance.

of a. That is, the existing test cases can be used to seed an automated search for additional test cases that also satisfy the testing criteria. This chapter considers two search algorithms, a hill climbing algorithm and an estimation of distribution algorithm, and compares them to unguided random test data generation. Hill climbing is a local search technique that will seek solutions near to the existing test cases in the space of possible test inputs. EDA (Estimation of Distribution Algorithm) is a global search technique that will consider solutions that are sampled from probabilistic distributions centered around the existing test cases.

4.4.1 Combined Reduction & Generation Strategy

Figure 4.3 shows how test data generation techniques work in conjunction with test suite minimisation techniques in order to enhance low latency of a test suite. A test suite minimisation technique selects the best subset according to a quality metric μ, which is set to branch coverage in the empirical study present in the chapter. This forces the enhancement strategy to 'work harder' because branch coverage subsumes statement coverage. These selected *model* test cases are fed into a test data generation technique, which seeks to generate new test cases that are different from the model test cases but still achieve the same quality metric value. The generated test cases are then added to the test suite for the next iteration.

Both the hill climbing algorithm and EDA work in conjunction with the additional greedy algorithm in order to repair a low latency test suite. Additional greedy algorithm selects the best subset according to a quality metrics μ, which is set to code coverage in the feasibility study present in the chapter. These *model* test cases are fed into test data generation techniques; the hill climbing algorithm and the estimation of distribution algorithm try to generate new test cases that are different from the model test cases, but still achieves the same quality metrics. The generated test cases are then added back to the test suite for the next iteration.

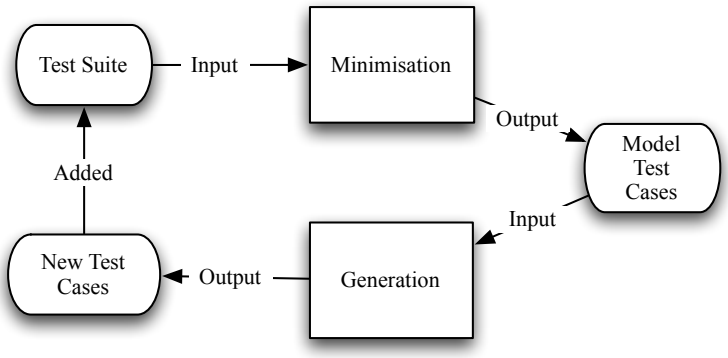

Figure 4.3: Latency Enhancement Overview

A single fitness function is used to guide both search algorithms, facilitating comparison. Let t be the individual test case the fitness of which is being measured, and t' be the original test case for which t seeks to mimic the behaviour. Let $\Delta_\mu(t, t')$ be the difference in quality metrics between two test cases, and $\Delta_d(t, t')$ be the distance between the input vectors of two test cases.

$$\Delta_\mu(t, t') = |\mu(\{t\}) - \mu(\{t'\})|$$

$$\Delta_d(t, t') = distance(t, t')$$

The fitness value of the test case t, $f(t)$, is defined as follows:

$$f(t) = \begin{cases} \Delta_d(t, t') & \text{if } \Delta_\mu(t, t') = 0 \wedge \Delta_d(t, t') > 0 \\ 0 & \text{if } \Delta_\mu(t, t') = 0 \wedge \Delta_d(t, t') = 0 \\ -\Delta_\mu(t, t') & \text{if } \Delta_\mu(t, t') > 0 \wedge \Delta_d(t, t') > 0 \end{cases}$$

If t is the same as t', the fitness function returns 0. However, if t is different from t' yet still achieves the same quality metric ($\Delta_\mu(t, t') = 0$), then t is guaranteed to receive a fitness value higher than 0, thereby encouraging the solution to move farther away from t'. Finally, if t is different from t' but has lower quality metric, t is guaranteed to receive

a fitness value lower than 0, but encouraged to reduce the difference in quality metric.

It should be noted that, with this particular search problem, finding the global optimum is not as important as finding as many qualifying solutions as possible. From the definition of the fitness function, qualifying solutions will be the test cases with fitness values higher than 3.

4.4.2 Hill Climbing

The hill climbing algorithm is one of the simplest local search algorithms [180], which is shown in Algorithm 2. The algorithm requires a definition of 'neighbouring solutions' for a given problem. The algorithm starts the search from a random solution, and considers neighbouring solutions. If a neighbour has higher fitness than the current position, the algorithm *climbs* to the fitter neighbour. This is repeated until there is no fitter neighbour, at which point the algorithm has reached one of the local optima in the search space. The *steepest ascent* hill climbing algorithm moves to the neighbour with the highest fitness, whereas the *first ascent* hill climbing algorithm moves to the first neighbour it considers with higher fitness than the current solution.

Algorithm 2: Hill Climbing Algorithm

```
(1)  x ← a random solution
(2)  while true
(3)      N ← neighbours of x
(4)      if ∃x′ s.t. x′ ∈ N ∧ fitness(x′) > fitness(x)
(5)          x ← x′
(6)      else
(7)          break
(8)  return x
```

For the latency enhancement study, the algorithm is modified to start from the known test case. Neighbouring solutions are defined as test cases that contain a single input variable that differs from the known test cases. These are created by either adding or subtracting a predefined amount to each input variable of the known test case, which

results in $2n$ neighbouring solutions for a test case with an input vector of length n.

One way of escaping local optima is to restart the algorithm with a different starting solution whenever the algorithm reaches a local optimum. However, in the version used in the study, the starting solution is set to the known test case in order to explore the input space near the known test case. To avoid deterministic behaviour, the algorithm used in the study adopts a *random first ascent*; the algorithm considers the neighbouring solutions in random order and moves to the first neighbour with higher fitness, which introduces randomness to the algorithm. Since the goal of the search is to explore the search space around the known test cases rather than finding the global optimum, the inability to escape local optima is thought to be less critical so long as the algorithm retains the ability to explore the neighbouring solutions.

4.4.3 Estimation of Distribution Algorithm

EDAs (Estimation of Distribution Algorithms) were first introduced in the field by Mühlenbein and Paaß [146]. They have been previously applied to software test data generation [185, 186] by Sagarna et al. The biggest difference between EDA and other evolutionary computation heuristics is that EDA does not rely on cross-over and mutation operators in order to generate new individual solutions. Instead, the individual solutions forming the population of the next generation are generated from a sampling of probability distribution, estimated from the previous generation.

Algorithm 3: Estimation of Distribution Algorithm

(1) $P_0 \leftarrow$ generate initial population randomly
(2) **while** stopping criterion is not met
(3) $\quad S_{i-1} \leftarrow$ select fitter individuals from P_{i-1}
(4) $\quad E_{i-1} \leftarrow$ probability distribution of S_{i-1}
(5) $\quad P_i \leftarrow$ sample E_{i-1}

Algorithm 3 shows the top level view of EDA. The algorithm starts by randomly generating the individual solutions that form the initial population. During the following

iterations, the algorithm first selects fitter individual solutions, guided by a predefined fitness function. Based on the selected individual solutions, the algorithm then estimates the probability distribution of the individuals. The next generation of individual solutions are sampled from the estimated probability distribution.

As in the case with the hill climbing algorithm, the estimation of distribution algorithm is modified in order to benefit from the knowledge of existing test cases. The initial population is not generated randomly; instead, Gaussian distributions are formed around the input variables of the known test cases (with mean values equal to the known values) and the initial population is sampled from these Gaussian distributions. By controlling the variance in the Gaussian distributions, it is possible to set the range of exploration around the known test cases.

4.5 Enhancement & Efficiency Study

The research questions for the enhancement and efficiency empirical study are: **RQ3. Enhancing Latency:** Can latency be improved by algorithms that implement the proposed combination of minimisation and generation?

RQ4. Assessing Efficiency: How efficient is the proposed approach to enhancing latency of test suites? Which search algorithm performs best for the proposed approach?

4.5.1 Subject Programs

A set of well-known benchmark programs for structural test data generation techniques is used. These are described in Table 4.3.

`Triangle1` is an implementation of the widely used program that determines whether the given three numeric values, each representing the length of a segment, can form a triangle. `Triangle1` is used by Michael and McGraw in their study of test data generation [140]. `Triangle2` is an alternative implementation of the same program by Sthamer

Program	Branches	Search Space
`triangle1`	20	2^{96}
`triangle2`	26	2^{96}
`remainder`	18	2^{64}
`complexbranch`	22	2^{192}

Table 4.3: Subject programs for the latency enhancement study

who also studied test data generation for `remainder`, a program that calculates the remainder of the division of two integer input [195]. Finally, `complexbranch` is a program specifically created as a challenge for test data generation techniques [211]. It contains several branches that are known to be hard to cover. For all programs, the search space is both large and non-trivial.

The initial test suites for the subject programs are generated such that 100% branch coverage is achieved. The initial test suites for the studied programs were generated by branch-by-branch approach. Programs were instrumented for the measurement of branch coverage. For each branch in the program, a single test case was generated to make the predicates both `true` and `false`. This is standard practice in search based test data generation [139].

4.5.2 Experimental Design

The comparison between the hill climbing algorithm and EDA is performed by allowing both algorithms 3,000 fitness evaluations. Both algorithms are also compared to random test data generation technique, which generates 3,000 random test cases. All three algorithms are executed 30 times to factor out their inherent stochastic properties.

The fitness function described in Section 4.4.1 is used for both algorithms in order to facilitate comparison. The distance in quality metric for a single test case is measured by the Hamming distance between two binary strings that correspond to the code coverage of each test case. The distance between two test cases is measured by simply taking

the Euclidean distance between two input vectors, which are numeric for all programs studied.

Following Korel [105], the neighbours in the hill climbing algorithm are generated by adding and subtracting 1 to each input variable in the current test case. Since the fitness function encourages the solution to move farther away from the original test case, it is possible that the algorithm exhausts the given fitness evaluations while moving arbitrarily farther away from the original test case. Therefore, the search is restricted to 10 ascents from the original test case.

The maximum number of ascents that the hill climbing algorithm can take for a single test case is limited to 10 times, because it is possible for the algorithm to exhaust all the given fitness evaluations for the generation of a single test case for which arbitrarily large number of ascents is allowed. For example, if the algorithm is considering a test case (x, y) which makes the predicate `(x == 10)` to be `true` and the original test case is $(10, 0)$, it is possible to increase the fitness value by moving y away from 0 to either -2^{31} or $2^{31} - 1$, exhausting the fitness evaluations on the way. The restriction is applied only for the comparison of different algorithms.

For the initial Gaussian distributions that are used for the initialisation of EDA, the standard deviation values are set to 3 for all input variables. The population size is set to 30. The algorithm terminates under two conclusions: 1) the population converges; or 2) the total number of generations processed exceeds 100.

4.5.3 Results and Analysis

Figure 4.4 shows the result of the attempts to enhance low latency of test suites by generating additional test cases with the algorithms described above. Each algorithm is executed 30 times, resulting in 30 different enhanced test suites. For each enhanced test suite, the additional greedy algorithm is applied repeatedly in order to measure the quality metric (the branch coverage), with the average plotted in Figure 4.4. It should

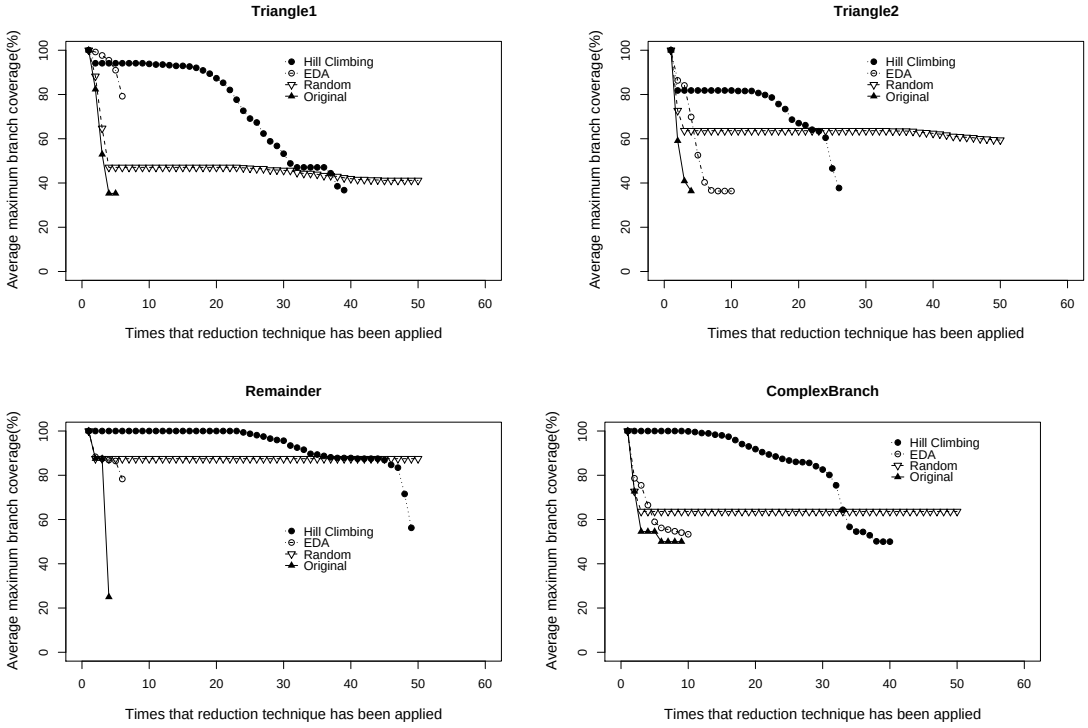

Figure 4.4: Results of latency enhancement : Each algorithm is given an identical budget of fitness evaluations for fair comparison of results. This budget allows the random search to produce suites that allow more minimisations to be made for a given budget. However, as can be seen, these suites are not as latent as those produced by the Hill Climber. The EDA can sometime outperform both Hill Climber and EDA in the coverage achieved by the first few minimisations, but it is the most expensive of the approaches, and the latency of its suites diminishes most rapidly.

be noted that the latency rates, $\lambda(coverage)(0)(greedy)$, are different at each execution of EDA and the hill climbing algorithm due to their inherent stochastic property. The results plotted in Figure 4.4 include only up to $min(\lambda(coverage)(0)(greedy))$ minimisations indicating worst case performance. On the other hand, the results for the random test data generation are plotted only up to 50 minimisations. The solid line with ▲ represents the latency of the original test suites.

The results from the hill climbing algorithm show that the enhancement strategy has improved the latency for all four programs, which answers **RQ3**. For `remainder` and `complexbranch`, the hill climbing algorithm manages to enhance the latency of test suites so that 100% branch coverage is maintained across consecutive minimisations. For `triangle1` and `triangle2`, the enhanced test suites fail to maintain 100% branch coverage but their coverage drops much more slowly than both that of the original test suite and those enhanced by other algorithms.

The reason why the hill climbing algorithm cannot maintain 100% branch coverage for `triangle1` and `triangle2` can be found in the semantics of the programs. The program `triangle1` contains a branch that determines whether the given 3 integers form an equilateral triangle. The definition of the neighbours used in the hill climbing algorithm prevents any generation of fitter neighbour for this branch. This is because the algorithm changes only a single input at a time to generate neighbouring solutions, preventing itself from reproducing a test case that corresponds to an equilateral triangle from another. However, from the original test case that forms an equilateral triangle, EDA generates three Gaussian distributions with the same mean value, which results in high probability of generating another test case that forms an equilateral triangle.

The program `triangle2` contains, apart from the branch that determines an equilateral triangle, branches that determine whether the triangle is a right-angled triangle or not. Both algorithms will explore around the original test case (a, b, c) such that $a^2 + b^2 = c^2$. Neither the Gaussian distributions nor the neighbourhood definition used

with the hill climbing algorithm is appropriate for finding an alternative test case that satisfies the condition.

The results for the program `remainder` are interesting because the random test data generation manages to maintain quality metric of 87.5% branch coverage. The remaining 12.5% branch coverage is controlled by two different predicates in the program that require either of the two input variable (a, b) to be equal to 0. The hill climbing algorithm shows the most successful performance because the neighbours are generated by changing a single input variable at a time, retaining the critical input value. On the other hand, the results from EDA show that the algorithm fails to retain the critical input value.

It is interesting to observe that the result of the random test data generation technique forms a flat plateau at different levels for different programs. This confirms findings of previous studies that show random test data generation can achieve a certain level of coverage, but cannot exceed this [139]. It should be also noted that, being a population-based evolutionary algorithm, EDA spends much larger amount of fitness evaluations in order to produce a single test case. This explains why the plotted lines for EDA are much shorter than those of other algorithms.

Program	$\bar{\lambda}_{random}$	σ_{random}	$\bar{\lambda}_{EDA}$	σ_{EDA}	$\bar{\lambda}_{HC}$	σ_{HC}	p_{H1}	p_{H2}	p_{H3}
`triangle1`	1.00	0.00	4.57	1.01	18.37	3.56	< 2.2e-16	< 2.2e-16	< 2.2e-16
`triangle2`	1.00	0.00	1.00	0.00	1.00	0.00	N/A	N/A	N/A
`remainder`	1.00	0.00	1.13	0.35	32.60	3.89	< 2.2e-16	0.0217	< 2.2e-16
`complexbranch`	1.00	0.00	1.00	0.00	20.67	2.66	< 2.2e-16	< 2.2e-16	< 2.2e-16

Table 4.4: Statistical analysis of latency enhancement strategy : The results of this analysis confirm statistically what can be seen visually in Figure 4.4; that the Hill Climbing is the most effective at enhancing the latency of test suites to which it was applied.

Table 4.4 shows the statistical analysis of the results shown in Figure 4.4. For each enhanced test suite s, the latency of enhanced test suites are evaluated by measuring the latency level $\lambda(coverage)(0.90)(greedy)s$, which means the largest number of times

the additional greedy algorithm can be applied to the enhanced test suites before the branch coverage of the outcome of the additional greedy algorithm falls below 90%.

The λ levels of three algorithms are compared pair-wise using one-tailed Welch's t-test with the significance level of 95%. Welch's t-test is an adaptation of t-test for two samples having different variance values. It tests the null hypothesis that the means of two normally distributed groups are equal. In the context of the study, the null hypothesis is that the λ levels achieved by different algorithms are are equal to each other. Three alternative hypothesis are formed as follows: H1. λ levels of EDA are greater than those of the random test data generation; H2. λ levels of the hill climbing algorithm are greater than those of the random test data generation; and H3. λ levels of the hill climbing algorithm are greater than those of EDA.

Apart from `triangle2` for which all three algorithms fail to increase λ above 1.0, the observed p-values for both H2 and H3 are significant at the 95% confidence level, confirming the alternative hypothesis that the λ levels of the hill climbing algorithm are higher than those of the random test data generation. The observed p-values for H1 for `triangle1` and `remainder` are also significant at the 95% confidence level, confirming the alternative hypothesis that the λ levels of EDA are higher than those of the random test data generation. Overall, the statistical analysis of the results suggests that the hill climbing algorithm is the most effective algorithm among the three algorithms studied. This answers **RQ4**.

4.6 Threats to Validity

Threats to internal validity concern the factors that might have affected both the empirical study of latency measurement (Section 4.3) and empirical study of enhancement strategy (Section 4.5). For the empirical study of latency measurement, one potential concern involves the accuracy of the instrumentation of the subject programs. To ad-

dress this, a well-tested and widely used open source profiler and compiler tool (`gcov` & `gcc` respectively) were used to collect the code coverage. The analysed test suites are retrieved from a well-managed and widely-used software archive [41]. For the empirical study of enhancement strategy, the potential internal threat comes from the selection and configuration of the search algorithms utilised in the chapter. Since the test data generation approach adopted in the study is a novel one, no algorithm is known to be effective for the problem. Both algorithms are chosen for their ability to search near the known solutions, and their strengths and weaknesses for certain types of branches are discussed in Section 4.5.3.

The empirical study uses Welch's t-test to evaluate different techniques. As described in Section 3.5.1, Welch's t-test assumes a parametric distribution of the samples. The exact distribution of the sample for this empirical study is not known; however, it will approximate a normal distribution with a sufficiently large sample size according to the central limit theorem [161].

Threats to external validity concern the conditions that limit generalisation from the result. The primary concern for the study is the representativeness of the programs used in the feasibility because they are small benchmark programs. However, the sizes of search space for the programs are large and nontrivial. The programs also have successfully served as benchmarks for previous work on structural test data generation techniques [140, 195, 211]. Additional research will address the concern by considering additional programs.

4.7 Conclusions

This chapter introduced the concept of latency in test suites, providing theoretical formulations and empirical results for latency measurements and enhancement. In order to enhance low latency of test suites, the chapter introduced a test data generation tech-

nique based on the exploration of the search space around existing selected test cases. The strategy combines test data generation techniques with test suite minimisation techniques. The enhancement strategy study performed on benchmark programs shows that the proposed approach is capable of improving the latency of test suites.

Chapter 5

Test Data Augmentation for On-Demand Regression Testing

5.1 Introduction

The latency enhancement strategy outlined in Chapter 4 proposes a novel approach to regression testing. The strategy essentially depends on a method of generating new test data from the known and existing data. If the cost of this method is low enough, we can envision an entirely new way of performing regression testing that is free from the woes of low latency: an *on-demand* regression testing. In on-demand regression testing, the tester does not maintain a large pool of test cases with the expectation that the redundancy in the test suite will help fault detection. Instead, the tester keeps a small but essential set of test cases that individually satisfy one or a few of the testing goals. An iteration of the overall regression testing process is performed by generating a new set of test cases from this core pool of test data whenever required. In other words, the tester keeps a small number of high-value test cases as templates, which are then used to generate actual test data for regression testing.

This approach eliminates the need for the redundancy in the managed test suite,

while still preserving high level of latency throughout multiple iterations of regression testing. In fact, the latency of the on-demand test suite is only limited by the size of valid input space defined by the semantics of the SUT. The proposed approach does not necessarily have to replace the current approach of maintaining a test suite across iterations; two approaches can be combined to allow the tester a mixture of known test data and novel data generated on-demand.

The idea of using existing test data in order to generate additional test data renders itself very well to search-based software testing, because most meta-heuristic algorithms that have been used for test data generation require one or more initial solutions to start the search from. While the idea of test data augmentation may apply to other forms of automated test data generation, this chapter focuses on search-based test data augmentation in order to introduce concrete algorithms for augmentation and to evaluate the proposed augmentation approach against existing approaches to automated 'from scratch' test data generation. The chapter also focuses on structural test adequacy, though, once again, this focus serves merely to allow us to introduce concrete instantiations of the idea of test data augmentation; the existing structurally adequate test data is augmented with additional structurally adequate test data.

Like other forms of automated test data generation, current approaches to search-based test data generation start from a blank sheet of paper. That is, they assume that no test cases exist. Based on the execution of an instrumented version of the program under test, structural search-based testing seeks to find test cases that cover certain paths of interest (as measured by the instrumentation). The measurement of the difference between a path traversed and a desired path provide the fitness function value that is used to guide the search. As well as structural testing [68, 75, 98, 138], search-based approaches have been applied to functional testing [203, 206], and verification of non-functional properties [157, 212, 213]. However, in every case, the search starts from scratch, assuming no previous test data exist. Where there are already test data, these

might be exploited with the potential to both reduce subsequent test data generation effort and to improve its effectiveness. It is this potential of existing test data that is explored in this chapter.

This chapter introduces test data augmentation, a technique that generates test data by exploiting the existing test data. In the context of regression testing, the term *augmentation* was used to represent the problem of generating additional test cases that will test the newly added features of the System Under Test (SUT) [79]. Here, we use the term simply to represent the process of adding additional test cases to an existing test suite to improve its fault detection capability, without assuming any change to SUT. But why do we want to generate additional test cases if there already exists a test suite? There can be various reasons. Since testing can only reveal existence of faults and not the lack of them, executing additional test cases can only increase the confidence in the program under test. Empirical research in test case management techniques has also shown that it is desirable to have certain level of redundancy in testing [165, 166, 176]. Additional test suites can be also beneficial for verification of fault fixes or statistical estimation of program reliability.

The proposed test data augmentation technique uses meta-heuristic search to search for a test case that behaves in the same way as, but with different test input from, the original test case. This is achieved by applying a series of a pre-defined set of modification operators to the original test case. Each modification operator applies some changes to the original test case while preserving a certain aspect of the original test case. The new test case is included in the new test suite if it shows the same behaviour as the original test case. Here the same behaviour means the same contribution to test objectives. For example, for structural testing the new and modified test case should also achieve the same coverage as the original test case.

This chapter introduces a novel search-based test data augmentation algorithm and presents the results of an empirical comparison with one of the state-of-art test data

generation techniques. Both the efficiency and the effectiveness of two techniques are evaluated. The results are promising; savings in the cost as large as two orders of magnitude can be observed. With a suitable configuration, test data augmentation can generate an additional test suite that achieves the same structural coverage as the original test suite. Furthermore, having the new test suites generated by test data augmentation technique proves to be beneficial both quantitatively and qualitatively. First, it tends to allow for higher mutation score, showing the benefits of having additional test suites. More interestingly, the mutation faults detected by test data augmentation technique are different from those that are detected by the original test suite and the ones generated by a state-of-art search-based test data generation technique; the details of the results from mutation testing are discussed in Secton 5.6.3.

The rest of the chapter[1] is organised as follows. Section 5.2 discusses background material on test data generation. Section 5.3 presents the motivation for test data augmentation and the research questions. Section 5.4 illustrates the proposed search-based test data augmentation technique in detail. Section 5.5 describes experimental setup used for the empirical study. Section 5.6 presents the empirical evaluation of the proposed technique. Section 5.7 interprets the empirical results. Section 5.8 presents a case study with real world subjects. Section **??** discusses the potential threats to validity. Finally, Section 5.10 concludes and lists directions for future work.

5.2 Background

Automatic test data generation using meta-heuristic search techniques has been a productive area of research in recent years. Manual test data generation, sill widely used in the industry, is very costly and laborious. The application of meta-heuristic search

[1]This chapter is an extended version of the author's STVR paper: S. Yoo and M. Harman, Test Data Augmentation: Generating New Test Data from Existing Test Data. *Software Testing, Verification and Reliability, under revision.*

techniques to test data generation presents a promising possibility of automating the process. Various meta-heuristics including hill climbing, simulated annealing and genetic algorithms have been applied to testing problems with objectives such as structural coverage, specification-based testing, and non-functional properties [139].

```
(1)   read(x)
(2)   read(y)
(3)   if x == y then print x
(4)             else print x - y
```

Search-based test data generation is a form of dynamic testing. The program under test has to be instrumented according to the test objectives, e.g. structural coverage or execution time. The meta-heuristic search aims to find test data that meets the testing objectives by executing the instrumented program for the evaluation of the fitness of a candidate solution. The combination of program execution and search algorithm dates back to Miller and Spooner [141]. They produced a *straight* version of the program under test for each path, converting predicates to constraints. Solving these constraints produces a test input that executes the chosen path. Later, Korel extended the idea for Pascal programs using a hill climbing local search algorithm [105]. Korel formalised a concept called *branch distance*. Branch distance of a predicate measures how close it is to being evaluated as `true`. Once instrumented for branch distance, a branch can be resolved in a way that is desirable for the test objective by optimising the distance. For example, suppose that we want to execute the `true` branch of the predicate in line (3) of the following program. The branch distance for the predicate `x == y` is calculated as $|x - y|$. This forms the fitness function for a test input. That is, any test input to the program, $t = (x, y)$, is evaluated using $f(t) = |x - y|$ in order to reach the `true` branch in line (3). It is then possible to apply a variety of meta-heuristic optimisation techniques to minimise or maximise the fitness function. In the example program, minimising the

fitness function will produce a test input $t = (x, y)$ such that $|x - y| = 0$, i.e., the one that will execute the `true` branch.

The branch distance concept is a generic approach to structural testing and can be used with other meta-heuristic search techniques such as simulated annealing [204, 205], genetic algorithms [9, 20, 140, 155, 211], Estimation of Distribution Algorithm (EDA) and scatter search [185, 186]. Search-based test data generation has been applied to testing of non-functional properties such as worst-case execution time [157, 212] and dynamic memory consumption [114].

Several techniques were developed in order to overcome problems in search-based test data generation, e.g., large search space and challenging search landscape. Input domain reduction tries to reduce the size of the search space by putting constraints to test input using symbolic execution [148], or slicing the program under test so that irrelevant test inputs are removed [74]. Testability transformation tries to transform search landscape into one that is friendlier to automated test data generation [87, 109, 137]. For example, removing a flag variable can transform a large plateau into a landscape with gradient. Test data are generated from the transformed program, but applied to the original program to test it.

The idea of using existing test cases for the generation of new ones is not entirely new. Similar ideas have appeared previously in the literature. Baudry et al. presented the bacteriologic algorithm for test case optimisation [11–13]. The bacteriologic algorithm applies a series of mutations on an initial test suite, and incrementally evolves a test suite that is deemed to be superior to the original one in terms of mutation score. The algorithm is initialised with an existing test suite, in a manner similar to the test data augmentation technique proposed in this chapter. Tlili et al. improved Evolutionary Real-Time Testing (ERTT) by seeding the ERTT algorithm for the Worst-Case Execution Time analysis with a test suite that achieves high structural coverage [201]. Without the seeding approach, some parts of the source code were never executed during the

search. The seeding approach helped the ERTT algorithm to generate more reliable WCET solutions that execute the larger parts of the program under test. The differences between these algorithms and the technique proposed in this chapter will be discussed in Section 5.4.5.

5.3 Problem Statement

5.3.1 Motivations for Search-based Test Data Augmentation

This chapter presents a search-based test data generation that uses the knowledge of existing test data. We call the proposed approach test data *augmentation*; it aims to augment existing test data with alternatives derived from those already available. In many cases, it is not unrealistic to assume that a tester in a software organisation already has some test data. Why do we want more test data when we already have some? Since testing can only reveal the existence of a fault, not the lack of it, repeated testing can only raise the confidence in the correctness of the program under test. Having a low-cost method of generating additional test data from existing data can be beneficial in many ways. It may prevent the program under test becoming over-fitted to existing set of test data. It may be helpful when the tester wants to gain statistical confidence in the correctness of the program behaviour.

The knowledge of existing test data can make the generation of the new test data less expensive. Consider the simplified visualisation of a fitness landscape in Figure 5.1. The x-axis represents all the possible test data in the input domain; the y-axis depicts the possible fitness values for some unspecified test objective. The image in the left shows how the traditional search-based test data generation techniques work; they start with a random test data and try to find a qualifying solution by optimising on the fitness function. The image on the right shows how we can exploit the knowledge of existing test data to generate additional test data. Assuming that there exist a small *window* of

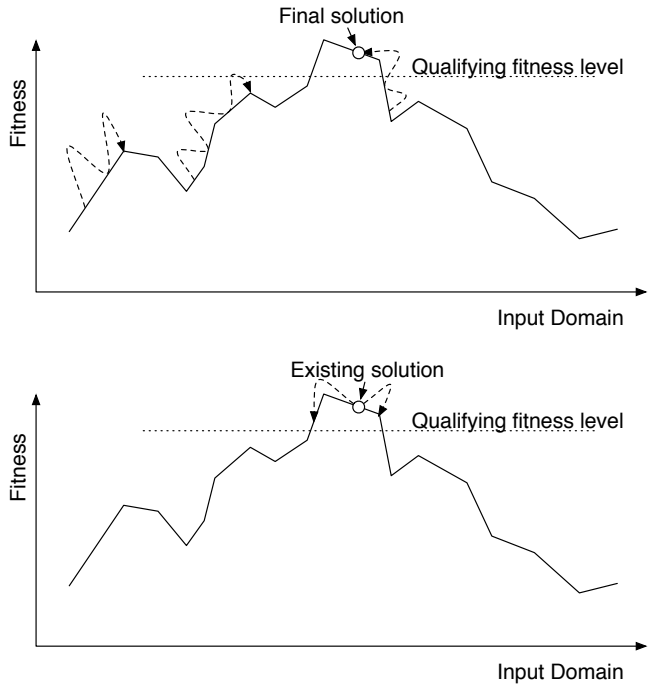

Figure 5.1: The top image illustrates the way in which traditional search-based test data generation techniques work. Meta-heuristic search techniques start from one or more random solutions, and try to obtain a qualifying solution by optimising the fitness function. The bottom image shows how the test data augmentation works. Since a qualifying solution is already known, meta-heuristic search is applied to explore regions of the search space that are close to the qualifying solution.

qualifying test data around the one that we already have found, we can expect to find additional test data, near the existing one, that also qualify.

A meta-heuristic search similar to the one depicted on the right side of Figure 5.1 has two major advantages compared to other meta-heuristic optimisation. The shape of search landscape becomes less problematic since the search starts from a point with sufficient fitness value. Similarly, the size of search landscape becomes less problematic since the search only needs to explore the regions of search space that are close to the existing solution.

This chapter presents a search-based test data augmentation technique, which for-

mulates test data augmentation as a search *around* the existing test data. In order to perform search-based test data augmentation, the following elements need to be defined:

- **Neighbourhood:** in order to search *near* the existing test data, we have to define what constitute the near-neighbours that we will search through.

- **Search Radius:** given a definition of near-neighbours, how far from the existing test data should we search?

Once the definition of near-neighbours and the search distance is fixed, the proposed search-based test data augmentation proceeds following the algorithm outlined in Algorithm 4. While Algorithm 4 is based on a local search technique, the same idea can be applied to other meta-heuristic search techniques.

Algorithm 4: Outline of search-based test data augmentation

```
(1)   currentSol = existingSol
(2)   while within the search radius
(3)       if near neighbours of currentSol contains a qualifying solution
          newSol
(4)           currentSol = newSol
(5)       else
(6)           break
(7)   return currentSol
```

5.3.2 Problem Statement

Depending on the program under test and the relevant test objective, there may or may not exist a 'window of qualification' as described in Section 5.3.1. The definition of being "near" to the existing test data also plays an important role in deciding whether such a window of qualification exists. In general, there is no particular reason to believe that it always exists. Indeed, a counter example can be found by considering the predicate

`x == 0`. Only the test data that has 0 as the value of `x` can qualify if we are trying to make the predicate `true`. Therefore, the thesis presents the proposed approach as a complimentary alternative to existing techniques, not as a replacement, and aims to observe the trade-off between two different approaches. Indeed, the proposed approach to test data generation assumes the existence of a known set of test data, which has to be generated using some other techniques including manual test data generation.

This chapter evaluates test data augmentation in terms of efficiency and effectiveness across different input domain, with comparison to Iguana, a state-of-the-art tool for search-based test data generation [136]. The first research question concerns the efficiency of the proposed technique. Efficiency is measured by counting the number of times that the program under test has to be executed in order to generate a set of test data that meets test objectives.

RQ1. Efficiency : How efficient is test data augmentation, compared to search-based test data generation, across different input domains?

The second and third research questions concern the effectiveness of test data augmentation, in terms of branch coverage and mutation score respectively.

RQ2. Coverage: How much branch coverage does test data augmentation technique achieve?

RQ3. Mutation Faults: How high a mutation score does test data augmentation technique achieve?

Finally, we parameterise the definition of neighbouring solutions, search radius, and the size of the input domain and observe how much impact they have on efficiency and effectiveness of the proposed approach.

RQ4. Settings: What variance in effectiveness and efficiency is observed for different set-up of neighbourhood and search radius?

5.4 Search-based Test Data Augmentation

For the introduction of the proposed test data augmentation technique and the first empirical study, we confine our discussion to test data augmentation of numerical test input composed as a vector of integers. This allows us to have clear and intuitive definitions. However, the proposed approach can be applied to other types of test input provided that suitable definitions of 'nearness' and 'search radius' can be constructed. The case study in Section 5.8 presents how search-based test data augmentation technique can be applied to programs that require more sophisticated input type such as strings and arrays.

The proposed search-based test data augmentation is based on the hill climbing algorithm, but there is one important difference between the proposed technique and the 'normal' hill climbing algorithm (and test data generation techniques that are based on hill climbing). The hill climbing algorithms adopt random restart in order to escape local optima; test data augmentation assumes that the existing test data belong to global optima, and, therefore, always starts from a global optimum that corresponds to the existing test data. Note that there may exist more than one globally optimal solution in a search-based test data generation problem. For example, suppose the goal of a given search-based test data generation problem is to cover the `true` branch of the predicate `x > 3`. Any value of `x` that is greater than 3 will cover the branch, and therefore will qualify as a globally optimal solution.

Since the lack of randomness can inhibit the variety of resulting solutions, the second

difference is introduced. In hill climbing algorithms, the move towards the next candidate solution (climb) can be made in a few different ways including *first-ascent*, where the algorithm moves to the first neighbouring solution that has higher fitness than the current solution, and *steepest-ascent*, where the algorithm moves to the neighbouring solution with highest fitness. However, both approaches behave deterministically when the starting point is fixed. Therefore, the proposed test data augmentation adopts *random-first-ascent*, where the algorithm examines the neighbouring solutions in a random order and moves to the first neighbouring solution with a higher fitness than the current solution.

5.4.1 Neighbouring Solutions and Interaction Level

In search-based test data generation, neighbouring solutions of a given solution are defined as the solutions that have the minimal difference from the original. For example, for integer input domain, neighbouring solution of an integer i is often defined to be $\{i-1, i+1\}$. This study formulates the process of obtaining neighbouring solutions as an application of a set of modification operators, which is a natural extension of existing methods.

This study utilises four modification operators. Following Korel, the first and second modification operators are defined as $\lambda x.x+1$ and $\lambda x.x-1$ [105]. When applied to integer values, these two operators generate test inputs that are *shifted* from the original. The third and the fourth modification operators are $\lambda x.x*2$ and $\lambda x.\lceil x/2 \rceil$. These operators generate test inputs that are *scaled* from the original.

Finally, we define *interaction level* as the number of variables that are modified at the same time. We borrow the concept of 'interaction' from interaction testing, which is a test for interaction of different configurations [34, 159]. If a modification operator is applied to a test input vector of size m with interaction level of i, it means all $\begin{pmatrix} m \\ i \end{pmatrix}$

possible combinations of variables will be modified by the operator at the same time, resulting in a neighbouring solution set of size no bigger than $\begin{pmatrix} m \\ i \end{pmatrix}$.

For example, suppose that an integer input vector (2, 3, 1) is modified with the described modification operators using interaction level of 2. With $\lambda x.x + 1$ we get $\{(3, 4, 1), (3, 3, 2), (2, 4, 2)\}$. With $\lambda x.x - 1$ we get $\{(1, 2, 1), (1, 3, 0), (2, 2, 0)\}$. Similarly, $\lambda x.x*2$ and $\lambda x.\lceil x/2 \rceil$ result in $\{(4, 6, 1), (4, 3, 2), (2, 6, 2)\}$ and $\{(1, 1, 1), (1, 3, 0), (2, 1, 0)\}$ respectively. The final neighbouring solution set is the union of four sets: $\{(3, 4, 1), (3, 3, 2), (2, 4, 2), (1, 2, 1), (1, 3, 0), (2, 2, 0), (4, 6, 1), (4, 3, 2), (2, 6, 2), (1, 1, 1), (2, 1, 0)\}$.

Modification operators and interaction level allows flexible approach towards the generation of neighbouring solutions. In existing search-based test data generation, the neighbouring solutions are explored to improve the fitness value. In test data augmentation, neighbouring solutions are generated from an existing solutions that already qualifies for the given objective. Therefore, the neighbouring solutions should aim to *preserve* certain properties found in the original solution. For example, consider a predicate `x * 2 = y` and a test input $(x, y) = (3, 6)$. Applying $\lambda x.x + 1$ and $\lambda x.x - 1$ with any interaction level results in different resolution of the predicate. However, applying $\lambda x.x * 2$ with interaction level of 2 results in $(x, y) = (6, 12)$, which still resolves the predicate as `true`.

From the combined use of modification operators and interaction level, we can draw an analogy between the test data augmentation technique and mutation testing. In mutation testing, the program under test is mutated to simulate actual faults. In test data augmentation, the processed is reversed by modifying the test cases rather than the program. The constraint for mutation testing is that the mutants should be executable, whereas the constraint for test data augmentation is that the new test case remains an eligible test case for the program.

5.4.2 Search Radius

Search radius is defined as the upper limit to the number of modifications that can be applied to the original test case. The number of modifications has to have an upper limit so that the cost of the search process can be controlled. For example, suppose we are modifying a test input vector $(x, y) = (2, 1)$ in order to resolve the predicate $x > y$ to `true`, with interaction level of 1. A successful chain of modifications might, for example, consist of applying $\lambda x.x + 1$ repeatedly to x until the value of x reaches the maximum possible value within the input domain, MAX. The algorithm will evaluate every candidate solution in $\{(x, 1)|2 \leq x \leq MAX\}$ by executing the program under test $MAX - 1$ times. This may be too costly. By having an upper limit to the number of modifications, it is possible to control the size of the search area, and thereby the cost of the search.

However, within the predefined search radius, the proposed test data augmentation technique is encouraged to move away from the original test data as far as possible. The intuitive underlying assumption is that, when the program under test has already been tested with the existing set of test data, the set of additional test data is most valuable when it is as different from the original set as possible, while still fulfilling the criteria. We call this the *distance-value* assumption.

5.4.3 Fitness Function

The proposed search-based test data augmentation technique uses a novel fitness function for test data generation. Since it assumes that an existing test input that meets the test objective is available, the new fitness function aims not to decrease the fitness below the qualifying level, rather than to increase the fitness above the qualifying level.

Let t be the individual test input, the fitness of which is being measured, and t' be the original test input known to meet the test objective. Let μ be the measurement of quality of testing in terms of meeting the test objective. Let $\Delta_\mu(t, t')$ be the difference

in quality metric between two test inputs:

$$\Delta_\mu(t, t') = |\mu(\{t\}) - \mu(\{t'\})|$$

Since this study only considers programs that require numeric test input vector, the Euclidian distance between vectors are used for Δ_d. Similarly $\Delta_d(t, t')$ be the distance between the input vectors of two test cases.

$$\Delta_d(t, t') = \text{the distance between } t \text{ and } t'$$

It may be difficult to measure the difference in μ quantitatively; this study utilises the hamming distance between two binary strings that represent the branch coverage record of t and t' for Δ_μ. Hamming distance is the number of bits that need to be flipped in order to transform one binary string into another. Here, lower hamming distance means that t and t' cover similar branches in the program under test. Without losing generality, we define the fitness function to be maximised, i.e. higher fitness values are better. Based on these, the fitness value of the test case t, $f(t)$, is defined as follows:

$$f(t) = \begin{cases} \Delta_d(t, t') & \text{if } \Delta_{f_R}(t, t') = 0 \wedge \Delta_d(t, t') > 0 \\ 0 & \text{if } \Delta_{f_R}(t, t') = 0 \wedge \Delta_d(t, t') = 0 \\ -\Delta_\mu(t, t') & \text{if } \Delta_{f_R}(t, t') > 0 \wedge \Delta_d(t, t') > 0 \end{cases}$$

If t is the same as t', the fitness function returns 0. However, if t is different from t' yet still achieves the same quality metric ($\Delta_\mu(t, t') = 0$), then t is guaranteed to receive a positive fitness value, guiding the search towards t. Because of $\Delta_d(t, t')$, the search is encouraged to move farther away from t'. Finally, if t is different from t' but has lower quality metric, t is guaranteed to receive a negative fitness value, thereby ensuring that the search never escapes the window of qualification.

It should be noted that, with this particular search problem, finding the global optimum is not as important as finding as many qualifying solutions as possible. From the definition of the fitness function, qualifying solutions will be the test inputs with positive fitness values.

Algorithm 5: Test data augmentation algorithm

Input: A test suite containing existing test inputs, S, the fitness function, f, search radius, r, and a set of modification operators, M, and the interaction level, i

Output: A new set of test inputs, S'

```
(1)   S' ← {}
(2)   foreach t ∈ S
(3)     finalSol ← null
(4)     currentSol ← t
(5)     count ← 0
(6)     while count < r
(7)       nextSol ← null
(8)       N ← GENERATENEIGHBOURS(M, currentSol)
(9)       while true
(10)        remove a randomly selected neighbour, n, from N
(11)        if f(n) > f(currentSol)
(12)          nextSol ← n
(13)          break
(14)        if size(N) == 0
(15)          nextSol ← null
(16)          break
(17)      if nextSol == null
(18)        break
(19)      else
(20)        currentSol ← nextSol
(21)        count ← count + 1
(22)    if finalSol ≠ t
(23)      S' ← S' ∪ {finalSol}
(24)  return S'
```

5.4.4 Algorithm

The pseudo-code of test data augmentation algorithm used in this chapter is shown in Algorithm 5. The main loop in line (2) iterates over each existing test input in the given

test suite, S. After initialisation in lines from (3) to (5), the inner loop in line (6) initiates the search process. In line (8), the algorithm generates the set of neighbouring solutions, N, by calling GenerateNeighbours($M, currentSol$). The loop in line (9) repeats until there is no neighbouring solution in N. If the algorithm finds a neighbouring solution with higher fitness value than $currentSol$, $finalSol$ is updated and the algorithm exits the loop. If N runs out of solutions, $nextSol$ is marked as $null$ and the loop in line (8) exits. If $nextSol$ equals $null$ in line (17), it means that none of the neighbouring solutions in N had higher fitness value than $currentSol$, in which case the loop in line (6) exits even if $count \geq r$. Otherwise, $currentSol$ is updated with the non-$null$ $nextSol$ and next move begins. The loop in line (6) will eventually exit when $count$ becomes equal to r. Finally, if the algorithm has found $finalSol$ that is not $null$, it is added to the new test suite, S'.

Algorithm 6: GenerateNeighbours() subroutine

Input: An existing test input vector, t, and a set of modification operators, M, and the interaction level, i
Output: A set of neighbouring solutions, N

```
(1)   N ← {}
(2)   foreach operator op in M
(3)     foreach each combination C of i variables out of t
(4)       newNeighbour ← copy(t)
(5)       foreach variable v in C
(6)         update newNeighbour by replacing the value of v with
            op(v)
(7)       N ← N ∪ {newNeighbour}
(8)   return N
```

The psuedo-code of the subroutine that generates neighbouring solutions is shown in Algorithm 6. The loop in line (2) iterates over all modification operators available. Each of these operators is applied to $\begin{pmatrix} m \\ i \end{pmatrix}$ combinations of input variables, which are stored in $newNeighbour$. The variables not in the given combination remain the same as t since $newNeighbour$ is initialised with t in line (4).

A precise formulation of average computational complexity for the algorithm is problematic, due to the inherent probabilistic nature of the algorithm. For example, it is not always possible to determine the probability of a neighbouring solution having a higher fitness value than the current solution. However, it is possible to obtain the exact upper bound for the worst-case execution time. Let S be the existing test suite, which contains test cases that are input vectors of size m. Let M be the set of modification operators, i the interaction level, and r the search radius. For each test case in S, the algorithm considers up to $\begin{pmatrix} m \\ i \end{pmatrix}$ neighbouring solutions, which can be repeated for r times at maximum. For each consideration, $|M|$ operators are applied. Therefore, the upper bound for the worst-case execution time is calculated as following:

$$O(|S| \cdot \begin{pmatrix} m \\ i \end{pmatrix} \cdot |M| \cdot r)$$

5.4.5 Differences to Existing Techniques

The idea of initialising a meta-heuristic algorithm with a set of known solutions is shared between the search-based test data augmentation technique and other existing techniques such as the bacteriologic algorithm by Baudry et al. [11–13] or the seeded Evolutionary Real-Time Testing [201]. However, these algorithms differ from the proposed technique in one important aspect, which is preservation of the behaviour of the original test data. The bacteriologic algorithm does not consider the behaviour of individual test cases, because its fitness function evaluates test cases only collectively. With the proposed test data augmentation technique, it is possible to generate a specific test cases that follows the exactly same execution trace of the original test case while still being different from the original. The seeded ERTT algorithm does not preserve the behaviour of individual test cases, because it uses one property (structural coverage) as a starting point to evolve and improve another (WCET). Unlike the test data augmentation technique, therefore,

the resulting test data are not guaranteed to achieve the same level of structural coverage.

Another difference is how these two algorithms generate candidate solutions. The bacteriologic algorithm only uses a syntax tree based mutation, and the seeded ERTT uses selection, crossover, and mutation operators. On the other hand, the test data augmentation technique uses a set of modification operators, which allows for more flexibility to tailor the augmentation process for the semantics of the program under test.

The overall approach advocated in this chapter is also very general, and is not constrained to any particular algorithm. That is, the chapter introduces the idea that testing does not start, *ab initio*, with no existing test data. There will often be some existing test inputs and it may make sense to start the test data generation process from any such initial cases.

5.5 Experimental Design

This section sets out the experimental design for the empirical study that evaluates the proposed test data augmentation technique.

5.5.1 Iguana : Hill Climbing Test Data Generation

The proposed test data augmentation technique is compared to a well known search-based test data generation tool called Iguana (hereby referred to IG) [136]. IG uses an advanced version of hill climbing called *alternating variable method*. Each input variable in the test input is taken and adjusted, while other variables are kept constant. First the algorithm performs what is called an *exploratory* phase, in which the algorithm makes a probe movement to neighbouring solutions. If one of the probe movements turns out to be successful, i.e. produces higher fitness, then the algorithm enters what is called a *pattern* phase. In the pattern phase, the previous move is repeated in the same direction (adding

or subtracting to the same variable) while doubling the distance in each iteration. If such a move produces a solution with lower fitness than current solution, then the algorithm switches back to exploratory phase with the next input variable. This method ensures that the algorithm reaches local or global optima quickly.

IG produces test suites that achieve branch coverage, i.e. every branch in the program under test is evaluated to both `true` and `false`. Given a program under test, it targets each branch in the program sequentially and tries to generate a single test input that executes the branch. If it cannot generate a test input within set number of fitness evaluations, the search is terminated. This study utilise the default maximum fitness evaluation setting, which is to spend 50,000 fitness evaluations maximum per branch.

It should be noted that IG also contains test data generation toolkits that are based on genetic algorithms. However, since the proposed test data augmentation is based on a hill climb algorithm, the comparison is made only to the hill climb based test data generation toolkit of IG.

5.5.2 Subject Programs

A set of well-known benchmark programs for structural test data generation techniques is used: two versions of the `triangle` program, `remainder` and `complexbranch`. Each program contains from 18 to 26 branches, which, though small, provides non-trivial branch coverage possibilities. These programs take 2 to 6 input variables. `Triangle1` is an implementation of the widely used program that determines whether the given three numeric values, each representing the length of a segment, can form a triangle. It is also the example program shown in Algorithm 7. `Triangle1` is used by Michael and McGraw in their study of test data generation [140]. `Triangle2` is an alternative implementation of the same program by Sthamer who also studied test data generation for `remainder`, a program that calculates the remainder of the division of the two integers input [195]. Finally, `complexbranch` is a program specifically created as a challenge for test data

generation techniques [211]. It contains several branches that are known to be hard to cover.

One of the subject programs, `remainder`, constantly caused IG to consume large enough memory to halt with 32bit integer input domain. This is not a weakness of search-based test data generation, but rather an outcome of internal design decisions within IG. As a result, experiments regarding `remainder` do not use 32bit integer input domain and only use 8bit and 16bit integer domain.

5.5.3 Input Domain

Since one of the restrictions inherent in existing test data generation techniques is the restriction in size of the input domain, the proposed test data augmentation technique is tested against different input domain sizes to see if it can cope better with a significantly large input domain than traditional techniques. Both IG and the proposed technique were executed for three different input domain sizes : 8bit integers, [-128, 127], 16bit integers, [-32,768, 32,767], and 32bit integers, [-2,147,483,648, 2,147,483,647]. Combined with the number of input variables for the subject programs, this results in search spaces ranging in sizes from 2^{24} to 2^{192}. Both techniques are evaluated in terms of efficiency and effectiveness against different input domains.

5.5.4 Original Test Suites and Mutation Faults

The initial test suites used by the test data augmentation technique have been manually generated for each subject program and each input domain so that 100% branch coverage is achieved. For each branch in the subject programs, a single test input was generated to make the predicates both `true` and `false`. This is standard practice in search-based test data generation [139].

Apart from branch coverage, mutation score was used as a measure of effectiveness. Mutation faults are based on the notion of mutation testing, where the adequacy of test

Mutation Operator	Description
AORB	Replace basic binary arithmetic ops.
AORS	Replace short-cut binary arithmetic ops.
AOIU	Insert basic arithmetic ops.
AOIS	Insert short-cut arithmetic ops.
AODU	Delete basic unary arithmetic ops.
AODS	Delete short-cut arithmetic ops.
ROR	Replace relational ops.
COR	Replace conditional ops.
COD	Delete unary conditional ops.
COI	Insert unary conditional ops.
LOR	Replace unary logic ops.
LOI	Insert unary logic ops.
LOD	Delete unary logic ops.
ASRS	Replace short-cut assignment ops.

Table 5.1: A list of mutation operators used in this chapter.

cases is evaluated by introducing simple syntactic modifications to the program [27]. If a test case reveals this modification, it *kills* the mutant program. The mutation score of a test case is the total number of mutation faults that the test case has killed. A test case with higher mutation score is assumed to have a higher chance of detecting real faults, which is observed in several empirical studies of mutation testing for procedural languages [62, 149].

A well known mutation testing tool, `muJava`, was applied to the subject programs [128]. The types of mutation operators used in the study are described in Table 5.1. Application of these mutation operators produced 203, 241, 324, and 499 mutants for `triangle1`, `triangle2`, `remainder`, and `complexbranch` respectively. Sets of test data generated by different techniques were analysed against these mutants.

It should be noted that the mutation faults studied in the study may contain equivalent mutants, i.e. mutants that are semantically identical to the original program. The equivalent mutants raise serious problems for mutation testing. However, since equivalent mutants cannot be killed by the original test suite nor enhanced test suites, they do

not affect the validity of the findings of the current experiment; their existence can only strengthen the null hypothesis.

5.5.5 Evaluations

To cater for the inherent randomness in both techniques, each individual experiment was repeated for 20 times. By default, the interaction level was set to 1, similar to the alternating variable method, while search radius was restricted to 10. **RQ1** is answered by comparing the average number of fitness evaluations required for the generation of a new set of test data between the two techniques across different input domains. **RQ2** and **RQ3** are answered by comparing the average branch coverage and average mutation score between the two techniques across different input domains.

RQ4 is answered by comparing efficiency and effectiveness measurements of the test data augmentation technique against itself using different settings. In the first set of experiments for **RQ4**, we change the interaction level from 1 to the size of the input vector, while keeping other experimental factors constant, and observe effectiveness and efficiency. In the second set of experiments for **RQ4**, we change the search radius from 1 to 10, while keeping other experimental factors constant, and make the same observation.

5.6 Results and Analysis

5.6.1 Efficiency Evaluation

Figure 5.2 shows the efficiency measurement of the IG test data generation technique and test data augmentation technique (hereby denoted as TA) against the subject programs. The x-axis represents different input domains. The y-axis, which is in logarithmic scale, represents average number of fitness evaluations required for the generation of a new set of test data for each program. In general, test data augmentation is not much worse than IG (`remainder`), or much more efficient than IG `triangle1`, `triangle2`,

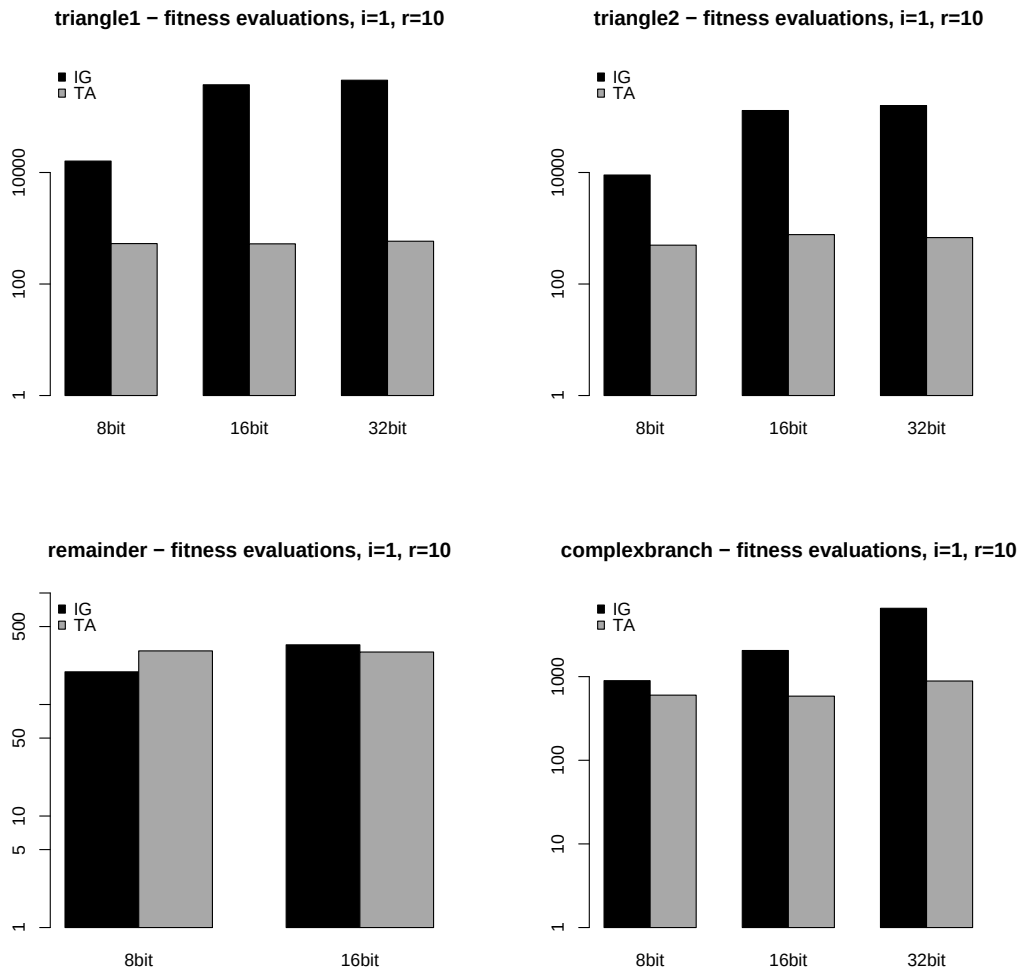

Figure 5.2: Comparisons of efficiency between test data augmentation and test data generation (using Iguana), across different input domains. The vertical axis is in logarithmic scale. Plots represent the average number of fitness evaluations required for the generation of a new set of test data. Apart from `remainder`, test data augmentation always requires a smaller number of fitness evaluations in order to generate a set of test data, which is statistically confirmed at 95% significance level. With `triangle1` and `triangle2`, the differences are more than two orders of magnitude.

Subject Programs	Input Domain	$\bar{n}_{IG}$	$\sigma^2_{n_{IG}}$	$\bar{n}_{TA}$	$\sigma^2_{n_{TA}}$	p-value
`triangle1`	8bit	16030.00	110408634.00	531.30	1618.54	$< 10^{-6}$
`triangle1`	16bit	373900.00	2043159037.00	525.30	766.85	$< 10^{-6}$
`triangle1`	32bit	452000.00	10559180.00	585.40	2132.99	$< 10^{-6}$
`triangle2`	8bit	9008.00	25917844.00	497.35	731.71	$< 10^{-6}$
`triangle2`	16bit	129200.00	695019728.00	769.10	769.10	$< 10^{-6}$
`triangle2`	32bit	158700.00	10490236.00	679.85	755.61	$< 10^{-6}$
`remainder`	8bit	196.30	1833.06	302.50	197.00	1.0
`remainder`	16bit	342.80	1119.36	295.55	163.94	0.0002
`complexbranch`	8bit	894.40	72117.63	600.60	1918.15	0.0006
`complexbranch`	16bit	2053.00	380509.10	584.30	1219.27	$< 10^{-6}$
`complexbranch`	32bit	6564.00	10241196.00	886.40	5452.04	$< 10^{-6}$

Table 5.2: Average number of fitness evaluations ($\bar{n}$) and variance (σ_n) for TA and IG.

and `complexbranch`). With `triangle1` and `triangle2`, the gain in efficiency is more than two orders of magnitude. More importantly, the gain increases as the input domain grows larger. While the test data augmentation technique shows relatively constant level of efficiency, test data generation technique does not cope well with larger input domains.

Table 5.2 presents the statistics observed in Figure 5.2 in more detail with statistical testing. Since there is no evidence to believe that the number of fitness evaluations required has a normal distribution, we use one-sided sign-test, which does not make assumptions about the probabilistic distribution of the population. The null hypothesis is that $\bar{n}_{IG}$ and $\bar{n}_{TA}$ has the same value. The alternative hypothesis is that $\bar{n}_{IG}$ is greater than $\bar{n}_{TA}$. The confidence level is 95%. For all cases except `remainder` with 8bit input domain, the resulting p-values indicates significant results. Therefore, we accept the alternative hypotheses for these cases. However, the alternative hypothesis is rejected for `remainder` with 8bit input domain. Therefore, we accept the null hypothesis. Overall, this answers **RQ1** positively. The statistical analysis shows that there is a significant gain in efficiency for most cases when test data augmentation is used. The amount of gain in efficiency can be as large as two orders of magnitude. It should also be noted that the cost of TA tends to be consistent across different input domains, whereas the cost

Subject Programs	Input Domain	$\bar{c}_{IG}$	$\sigma^2_{c_{IG}}$	$\bar{c}_{TA}$	$\sigma^2_{c_{TA}}$
`triangle1`	8bit	1.00	0.00	0.94	0.00
`triangle1`	16bit	0.91	0.00	0.94	0.00
`triangle1`	32bit	0.83	0.00	0.94	0.00
`triangle2`	8bit	1.00	0.00	0.82	0.00
`triangle2`	16bit	0.90	0.00	0.82	0.00
`triangle2`	32bit	0.86	0.00	0.82	0.00
`remainder`	8bit	1.00	0.00	1.00	0.00
`remainder`	16bit	1.00	0.00	1.00	0.00
`complexbranch`	8bit	1.00	0.00	1.00	0.00
`complexbranch`	16bit	1.00	0.00	1.00	0.00
`complexbranch`	32bit	0.98	0.00	1.00	0.00

Table 5.3: Average branch coverage ($\bar{c}$) and variance (σ_c) for TA and IG.

of IG tends to increase as the input domain grows larger. This is due to the fact that IG always has to cope with the whole input domain, starting with a random solution, whereas TA can focus on the small region around the original solution defined by the radius parameter.

5.6.2 Effectiveness Evaluation: Coverage

We turn to **RQ2** and investigate the branch coverage achieved by IG and TA. Figure 5.3 shows average branch coverage achieved by both techniques. The x-axis represents the different input domains. The y-axis represents average branch coverage achieved by both techniques across 20 executions. TA achieves 100% branch coverage for `remainder` and `complexbranch` for all input domains. However, it fails to achieve 100% branch coverage for `triangle1` and `triangle2`, although it achieves a constant level of coverage. IG achieves 100% branch coverage for all subject programs in 8bit input domain, but the coverage decreases as the input domain grows larger.

Table 5.3 presents the statistics observed in Figure 5.3 in more detail. Note that variances are all 0, meaning that all 20 executions achieved the same branch coverage with both techniques. Therefore, statistical hypothesis testing is not performed. For

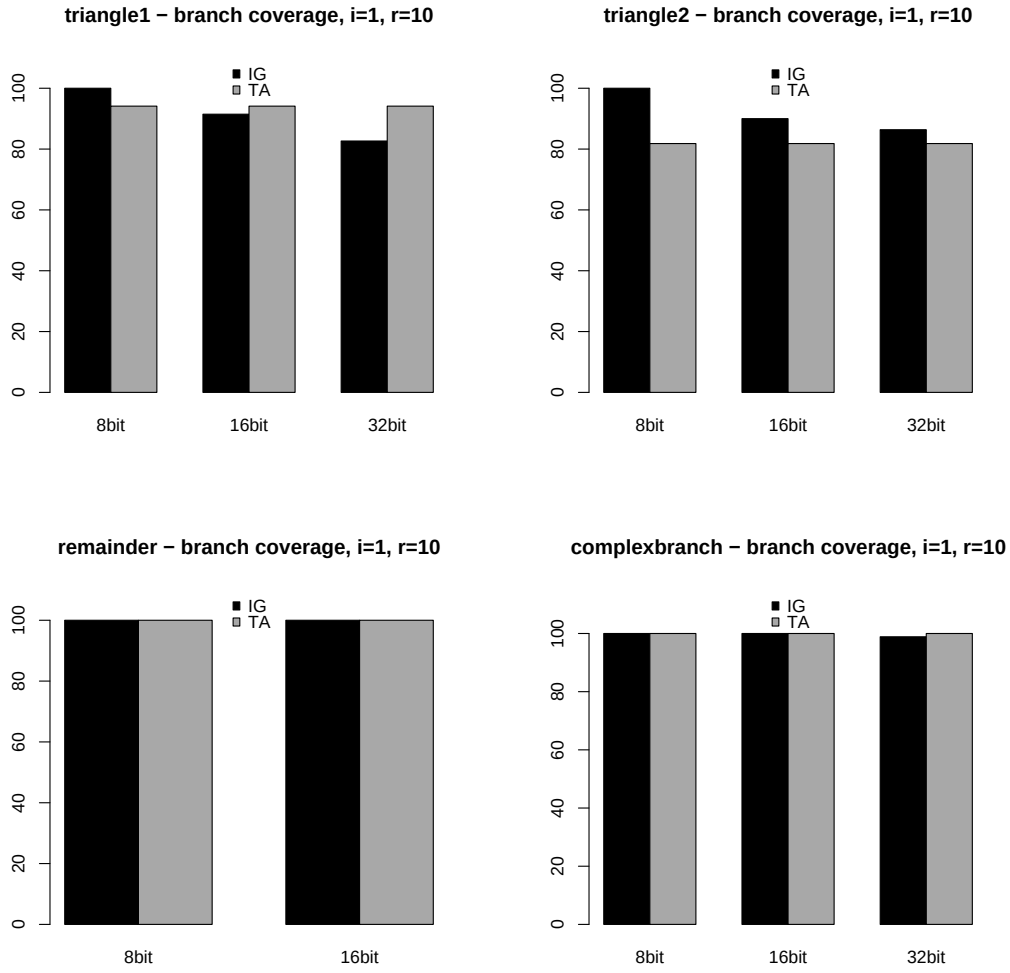

Figure 5.3: Comparisons of effectiveness between IG and TA, across different input domains, in terms of branch coverage. Plots represent average branch coverage achieved by both techniques. For `triangle1` and `triangle2`, IG shows decreasing branch coverage as the input domain grows larger, suggesting a possible lack of scalability. TA does not achieve 100% for these programs, but maintains the same branch coverage across different input domains. Both techniques achieve 100% branch coverage for `remainder` with all 20 executions. IG fails to achieve 100% branch coverage for `complexbranch` in 32bit integer domain. Note that the plotted coverage is the coverage achieved by the newly generated test suite alone; the original test suites were not included.

`triangle1` and `triangle2`, TA shows constant branch coverage for all input domains, whereas IG cannot maintain constant branch coverage as input domain grows. In fact, with the appropriate setting, TA can achieve 100% branch coverage (this will be discussed in Section 5.6.4). Both techniques achieve 100% branch coverage for `remainder` with all 20 executions. Both techniques achieve 100% branch coverage for `complexbranch` with the exception of IG with 32bit input domain. This provides a partially positive answer for **RQ2**. When the input domain is sufficiently large, TA can be as effective as, or more effective than IG, in terms of branch coverage. For smaller input domains, we still observe an attractive trade-off between efficiency and effectiveness.

5.6.3 Effectiveness Evaluation : Mutation Score

Figure 5.4 shows average mutation score achieved by both techniques. In order to justify the need for additional test suite, the mutation score of original test suites is also included. If executing the additional test suite increases the mutation score significantly, the cost of generating and executing the additional test suite may be justified. For `remainder` and `complexbranch`, both IG and TA mostly achieve either mutation score similar to the original, or higher than the original. The results for `triangle1` and `triangle2` form an interesting contrast. Note that for both programs, IG and TA failed to achieve full coverage as input domain grows larger. This has a significant impact on mutation score in case with `triangle2`. Lower coverage leads to lower mutation score because certain mutants are never covered. With `triangle1`, IG is also similarly affected by the lack of full coverage. However, TA shows relatively constant mutation score with `triangle1` across input domain even though it fails to achieve full coverage.

Table 5.4 presents the statistics observed in Figure 5.4. In order to compare the mutation score of IG and TA, hypothesis testing is performed. Without any assumption about the population distribution, the one-sided sign test is performed with 95% significance level. The p-value represents the results of sign test between m_{IG} and $\bar{m}_{TA}$.

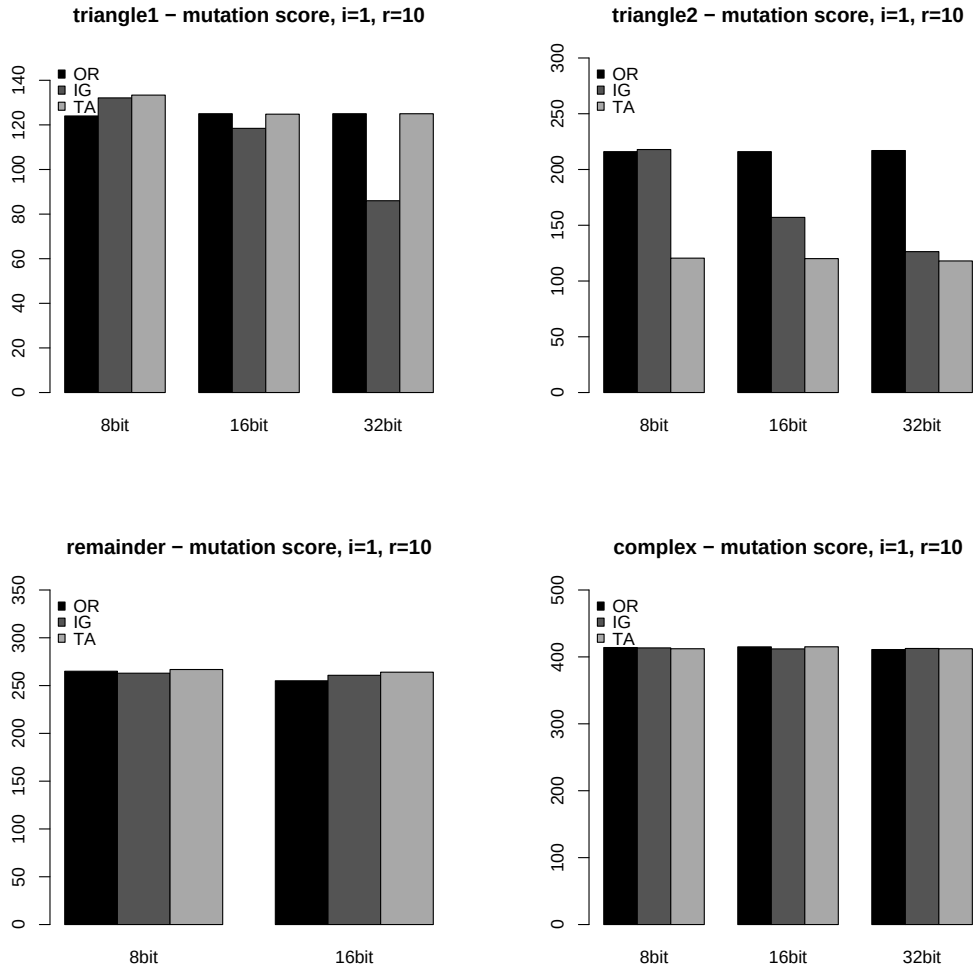

Figure 5.4: Comparisons of effectiveness between IG and TA across different input domains, in terms of mutation score. In order to test whether the additional set of test data improves the quality of testing, the mutation score of original test suites are included (OR). Plots represent either the mutation score (OR) or average mutation score (IG and TA). With `triangle2`, the lack of full coverage significantly affects both IG and TA. For other programs, TA is as effective as, or more effective than OR and IG in terms of mutation score.

Subject Programs	Input Domain	m_{OR}	$\bar{m}_{IG}$	$\sigma^2_{m_{IG}}$	$\bar{m}_{TA}$	$\sigma^2_{m_{TA}}$	p-value
`triangle1`	8bit	124	132.10	51.67	133.35	36.34	0.5000
`triangle1`	16bit	125	118.46	75.31	124.80	10.06	0.0898
`triangle1`	32bit	125	86.00	370.32	125.00	0.42	$< 10^{-6}$
`triangle2`	8bit	216	217.90	2.20	120.50	2.79	1.0000
`triangle2`	16bit	216	157.05	1158.79	120.10	3.67	1.0000
`triangle2`	32bit	217	126.35	0.56	118.00	1.58	1.0000
`remainder`	8bit	265	262.95	6.68	266.80	3.01	0.0037
`remainder`	16bit	255	260.75	5.99	264.00	0.00	0.0002
`complexbranch`	8bit	414	413.45	8.16	412.20	2.69	0.9283
`complexbranch`	16bit	415	411.95	7.63	415.10	0.62	$< 10^{-4}$
`complexbranch`	32bit	411	412.60	2.04	412.20	0.69	0.8950

Table 5.4: Average mutation score ($\bar{m}$) and variance (σ_m) for TA and IG. The column m_{OR} represents the mutation score achieved by the original test suite used by TA. The p-value represents the results of the one-sided sign test between m_{IG} and $\bar{m}_{TA}$. The null hypothesis is that $\bar{m}_{IG}$ is equal to $\bar{m}_{TA}$. The alternative hypothesis is that $\bar{m}_{TA}$ is greater than $\bar{m}_{IG}$. The alternative hypothesis is accepted for `triangle1` with 32bit input domain, `remainder` for all input domains, and `complexbranch` with 32bit input domain.

The null hypothesis is that $\bar{m}_{IG}$ is equal to $\bar{m}_{TA}$. The alternative hypothesis is that $\bar{m}_{TA}$ is greater than $\bar{m}_{IG}$. The result is mixed; none of the two techniques dominates the other across all experiments. The alternative hypothesis is accepted for `triangle1` with 32bit input domain, `remainder` for all input domains, and `complexbranch` with 32bit input domain. For other experiments, the results from two techniques either show no statistically significant difference or dominance by IG. However, considering the significantly lower cost of TA, this still provides an attractive trade-off between efficiency and effectiveness.

While mutation score is one possible measure of testing effectiveness, it does not consider the fact that different test suites kill different sets of mutation faults. In order to make a detailed comparison between the original test suite and the test suites generated by IG and TA, the mutation faults are classified according to the test suite that killed them. If, during the 20 executions, a mutation fault is killed by a test suite at least once,

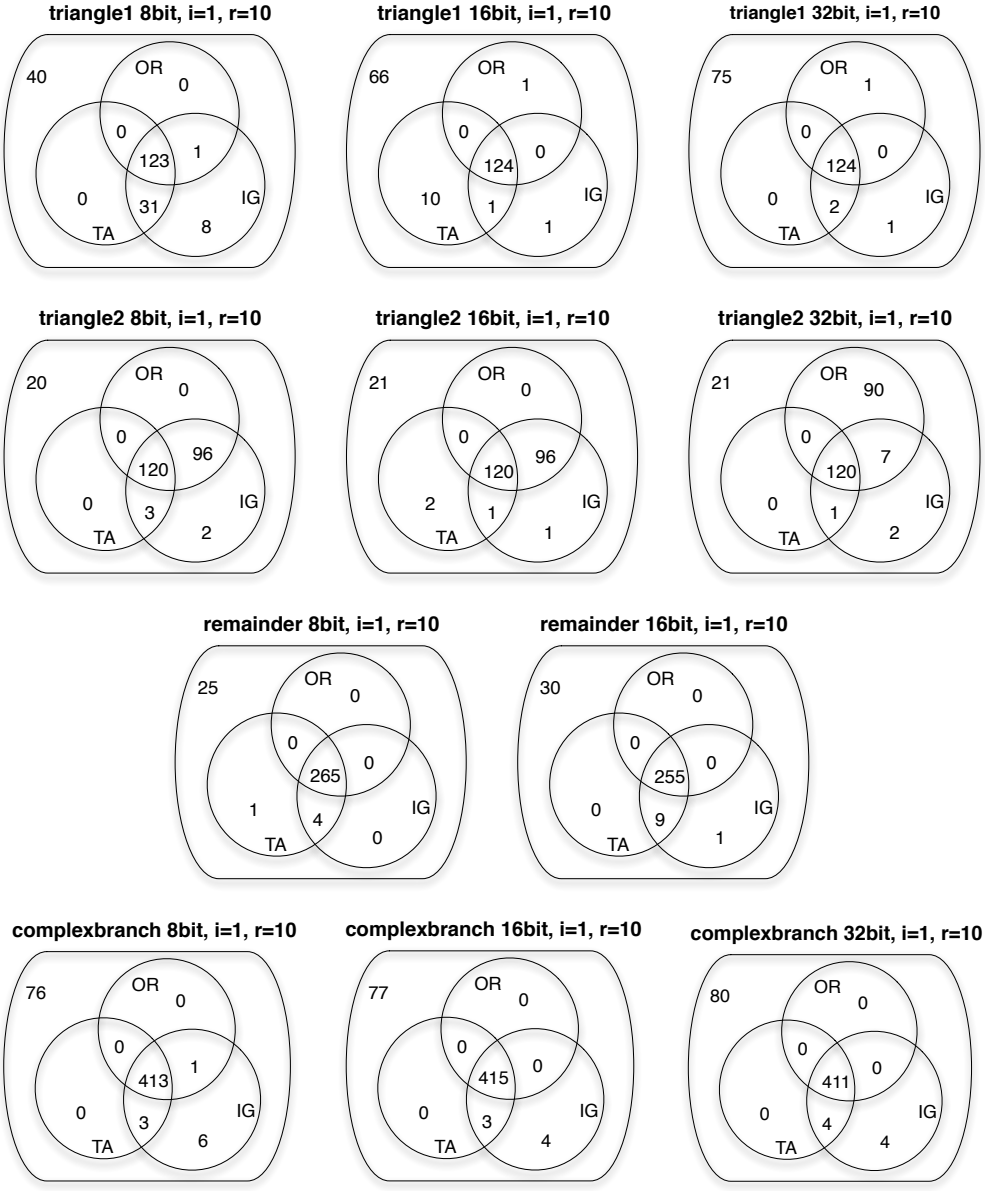

Figure 5.5: Venn diagrams classifying mutation faults according to the test suites that killed them. Note that for all experiments, $|TA - OR| > 0$, i.e. test data augmentation has made an improvement in mutation score for all experiments. IG has also made an improvement that is as good as or better than TA for all experiments. However, considering Section 5.6.1, TA still provides an attractive trade-off between efficiency and effectiveness.

the fault is classified as being killed by the test suite. This results in Venn diagrams shown in Figure 5.5. The set OR, TA, and IG represents the mutation faults that were killed by each technique respectively. Note that both $|TA - OR|$ and $|IG - OR|$ are greater than 0 for all experiments. Considering that the original test suites did achieve 100% branch coverage, this signifies that achieving structural coverage is not always a sufficient testing requirement. It therefore justifies the generation of an additional test suite, which TA can perform very efficiently. For `triangle2` with 8bit and 16bit input domains, TA fails to kill 96 mutation faults that are killed by OR and IG. This is due to the lack of complete coverage observed in Section 5.6.2. For all other experiments, TA kills at least more than half of the mutants in the set $IG - OR$. Indeed, $|TA - OR|$ is greater than $|IG - OR|$ for `triangle1` with 16bit integer input domain and `remainder` with 8bit integer input domain. Combining this with the gain in efficiency observed in Section 5.6.1, it provides an evidence that there exists an attractive trade-off between efficiency and effectiveness for TA. Overall this provides a positive answer to **RQ3**.

5.6.4 Settings: Impact of Interaction Level

Now we turn to the first part of **RQ4** and observe the impact of different interaction levels on efficiency and effectiveness. The minimum possible interaction level is 1; with 0 interaction level, it is not possible to generate a new test input. The maximum possible interaction level for `triangle1`, `triangle2`, `remainder` and `complexbranch` is 3, 3, 2, and 6 respectively, i.e. the size of test input vector for these programs. The input domain is fixed at 8bit integers. The search radius is fixed at 10.

Figure 5.6 shows the change in the average number of fitness evaluations against different interaction levels. The interaction level determines the number of neighbouring solutions that TA considers in a single iteration. With a test input vector of size m and interaction level i, the number of neighbouring solutions is bounded by $\begin{pmatrix} m \\ i \end{pmatrix}$. The

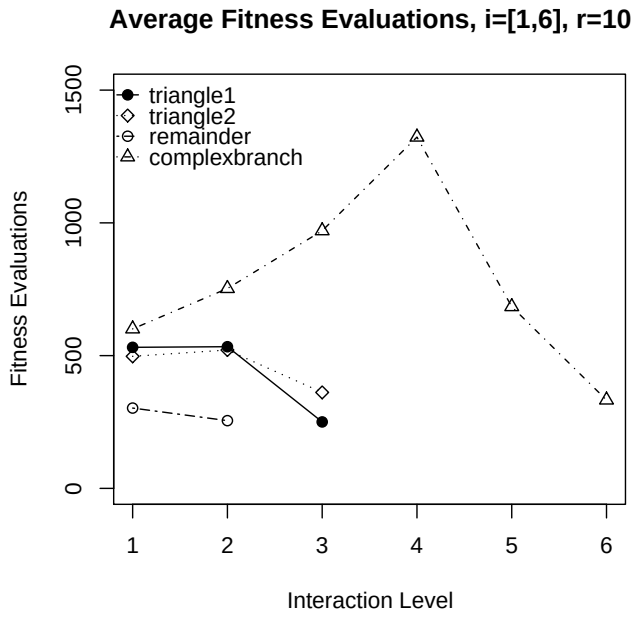

Figure 5.6: Plot of average number of fitness evaluations against interaction level. The number of neighbouring solutions is bounded depending on the size of test input vector and interaction level. A higher number of neighbouring solutions generally results in a higher number of fitness evaluations, i.e. more candidate solutions to evaluate. For `triangle1` and `triangle2`, $i = [1, 2]$ yields the most neighbouring solutions. For `complexbranch`, $i = 3$ yields the most neighbouring solutions, but $i = 4$ results in most fitness evaluations.

more neighbouring solution TA considers, the more fitness evaluations TA is likely to spend. Based on this, we expect each plot to peak at the point that corresponds to the pair of m and i that yields the maximum value $\begin{pmatrix} m \\ i \end{pmatrix}$:

- `triangle1` : interaction level $i = \{1, 2\}$ is expected to yields the most fitness evaluations.
- `triangle2` : interaction level $i = \{1, 2\}$ is expected to yields the most fitness evaluations.
- `remainder` : interaction level $i = 1$ is expected to yield more fitness evaluations than $i = 2$.
- `complexbranch` : interaction level $i = 3$ is expected to yield the most fitness evaluations.

The result shown in Figure 5.6 mostly confirms the expectations. For `triangle1`, `trioangle2`, and `remainder`, the expectations are confirmed. However, for `complexbranch`, the peak occurs at $i = 4$, not $i = 3$ as expected. This suggests that TA considered more candidate solutions when $i = 4$, even though at each iteration it generates more neighbouring solutions when $i = 3$. Therefore, the real impact comes not only from the number of maximum possible neighbours, but also from the number of those that are within the window of qualification. If the program under test allows sufficient number of neighbouring solutions that qualify, the number of fitness evaluation of TA is more likely to be limited by the search radius. On the other hand, if the program under test does not allow sufficient neighbouring solutions that qualify, TA will spend large number of fitness evaluations for disqualifying neighbouring solutions.

Figure 5.7 and Figure 5.8 observe effectiveness measures against interaction level in terms of branch coverage and mutation score respectively. In Figure 5.7, both `triangle1`

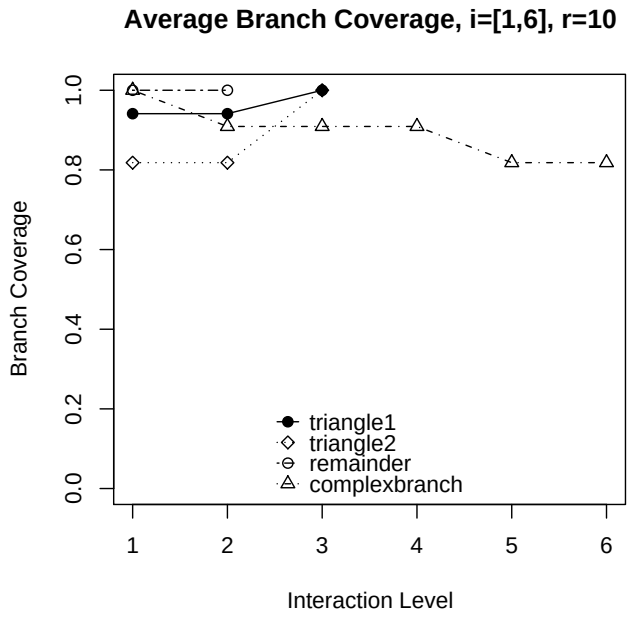

Figure 5.7: Plot of average branch coverage against interaction level. Both `triangle1` and `triangle2` achieve full coverage when $i = 3$ due to certain branches that require the interaction level of 3 in order to generate a qualifying solution. The branch coverage for `remainder` is not affected by increasing interaction level. For `complexbranch`, increasing interaction level actually reduces the average branch coverage achieved.

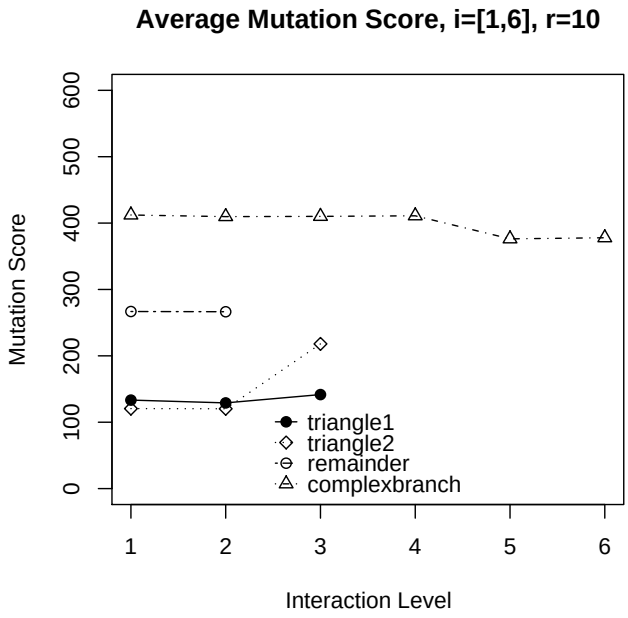

Figure 5.8: Plot of average mutation score against interaction level. Both `triangle1` and `triangle2` benefit from the increased branch coverage observed in Figure 5.7 when $i = 3$. On the other hand, `complexbranch` suffers from the decreased coverage as the interaction level increases. The mutation score of `remainder` is not affected.

and `triangle2` achieves full coverage when $i = 3$, complimenting the lack of full coverage observed in Section 5.6.2. More importantly, TA achieves full coverage with $i = 3$, while still remaining significantly more efficient than IG. Both programs contain branches that require the modification of all input variables in order to generate another qualifying solution. For `triangle1`, it is a branch that determines whether the given numbers form an equilateral triangle. From a set of three numbers that form an equilateral triangle, it is possible to generate alternative qualifying set of numbers when all three numbers are modified at the same time, by one of the modification operators $\{\lambda x.x + 1, \lambda x.x - 1, \lambda x.x * 2, \lambda x.\lceil x/2 \rceil\}$. Similarly, `triangle2` contains a branch that determines whether the given numbers form a right angle triangle. The generation of an alternative qualifying solutions is *guaranteed* only when $i = 3$ and only with the modification operator $\lambda x.x*2$. This provides a justification for utilising modification operators that are more complex than $\{\lambda x.x+1, \lambda x.x-1\}$. With domain knowledge, it may be possible to generate more complex but potentially more effective set of modification operators.

In Figure 5.8, it is possible to observe the positive impact of the increased branch coverage. Both `triangle1` and `triangle2` show improved mutation score with $i = 3$. The plot for `complexbranch` corresponds to the branch coverage of the program observed in Figure 5.7. With less coverage, TA kills fewer mutants as interaction level increases. For `remainder`, the increased interaction level has no significant impact on mutation score.

5.6.5 Settings: Impact of Search Radius

Finally, we turn to the second part of **RQ4** by observing the efficiency and effectiveness of TA while changing the search radius. The interaction level is fixed at 1, and the input domain is fixed at 8bit integers. TA is executed 20 times for each search radius value from 1 to 10.

Figure 5.9 plots the average number of fitness evaluations and average branch cov-

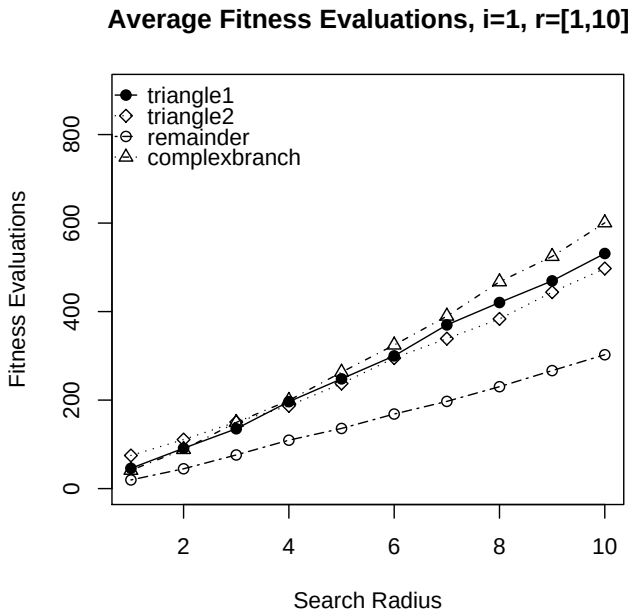

Figure 5.9: Plots of the average number of fitness evaluations against search radius. As can be seen, the average number of fitness evaluations shows a strong linear correlation to search radius; TA spends more fitness evaluations while trying to make more modifications.

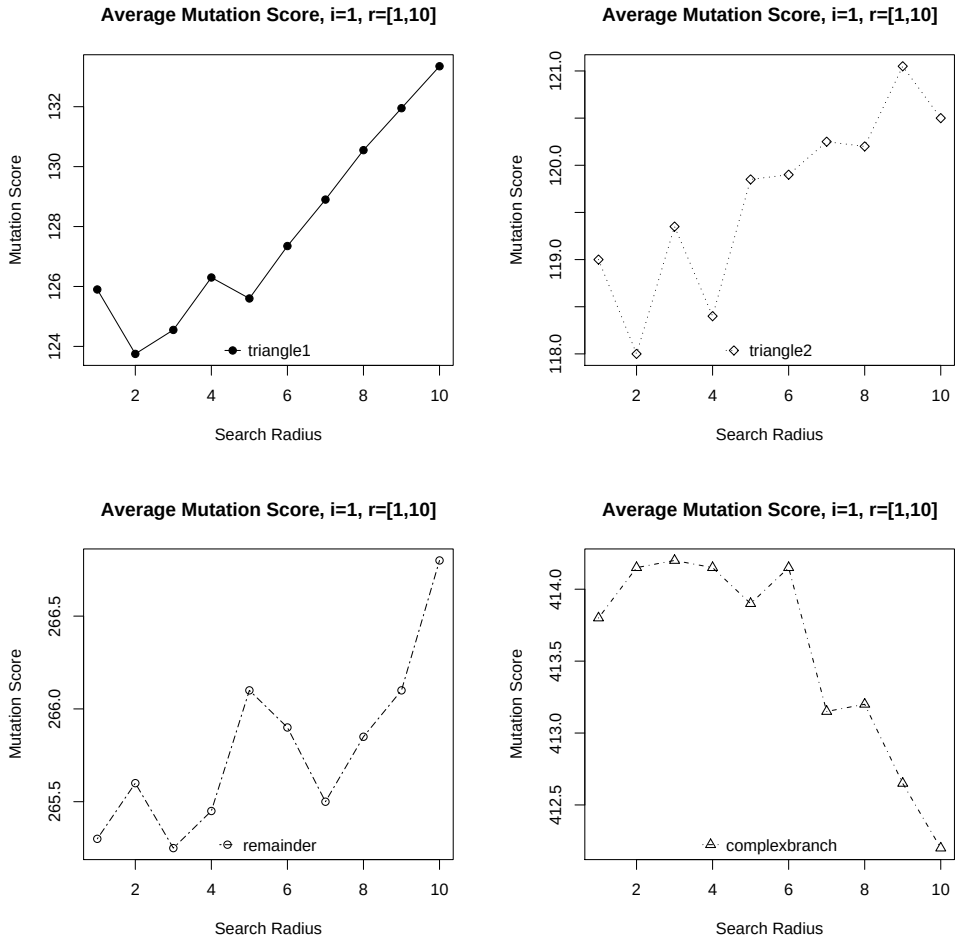

Figure 5.10: Plots of average mutation score against search radius. Except for `complexbranch`, average mutation score shows an increasing trend, providing a partial justification for TA to make as many modifications as possible. That is, the more an additional test case is different from the original, the more valuable it is for these programs.

Program	ρ_1	ρ_2
`triangle1`	0.9991983	0.9279003
`triangle2`	0.9977082	0.8569495
`remainder`	0.9992023	0.7774765
`complexbranch`	0.9983878	-0.8378015

Table 5.5: The column of ρ_1 shows the linear correlation coefficient between the search radius and the average number of fitness evaluation. All four programs show a very strong correlation. The column of ρ_2 shows the linear correlation coefficient between the search radius and the average mutation score. It shows a significant linear correlation for `triangle1`, `triangle2`, and `remainder`. On the other hand, `complexbranch` shows a negative correlation between the search radius and the average mutation score.

erage against search radius. For all subject programs, the average number of fitness evaluations shows a very strong correlation to search radius, which is expected. As the search radius increases, TA is allowed to make more modifications from the original test input, thereby spending more fitness evaluations. However, the observed coverage values remain constant at the maximum level that can be achieved for each program under the given configuration. TA achieved full branch coverage for `remainder` and `complexbranch`; it also covered all the branches in `triangle1` and `triangle2` except for the ones discussed in Section 5.7.

Figure 5.10 shows plots of average mutation score against search radius. With the exception of `complexbranch`, the observed trend is that average mutation score increases as search radius increases. This provides a partial evidence to confirm the distance-value assumption described in Section 5.4.2; that is, the more different an additional test input is from the original test input, the more valuable it is. Table 5.5 shows linear correlation coefficients between search radius and average fitness evaluations/average mutation score. The coefficients confirm the visual trends observed in Figure 5.9 and Figure 5.10.

5.7 Discussion

This section discusses factors that prevent IG from achieving full coverage for larger input domains. For `triangle1` and `triangle2`, IG fails to achieve full branch coverage when the input domain is set to 16bit or 32bit integers. This is due to the data dependency observed in both programs. The following is an excerpt from `triangle1`.

Algorithm 7: An implementation of `triangle` program

```
(1)   read(i, j, k)
(2)   tri = 0
(3)   if i == j then tri += 1
(4)   if i == k then tri += 2
(5)   if j == j then tri += 3
(6)   if tri == 0
(7)       if i + j ≤ k or j + k ≤ i or i +k ≤ j
(8)           tri = 4
(9)       else
(10)          tri = 1
(11)      return tri
(12)  ...
```

The program takes three integers as input, and classifies the input according to the type of triangle that can be formed with lengths of sides equal to the three integers. Suppose that we are trying to make the predicate in line (6) `false`, which means `tri != 0`. According to Korel's definition, the branch distance is calculated as $-|tri-0|$ [105], which should be minimised. The value of the variable `tri` is assigned between line (3) and (5) based on the values of the program input, `(i, j, k)`. Since the predicate in line (6) is control-independent from the predicates in line (3), (4), and (5), the branch distance of $-|tri - 0|$ cannot guide the input vector. That is, changes made to the input vector `(i, j, k)` do not correlate with the branch distance of the predicate in (6), except for the very rare cases where the algorithm starts with a random input vector that is very close to satisfying either one of the predicates in line (3), (4), and (5). This results in a

flat fitness landscape, thereby significantly weakening performance of any meta-heuristic search technique. As a result, IG often fails to cover the `false` edge of the predicate in line (6).

Compared to `triangle1`, `triangle2` contains additional branches that determine whether the given input forms a right-angle triangle. IG tends to fail to cover these branches as the input domain grows larger. There is no data dependency involved as in `triangle1`; the difficulty lies in the behaviour of alternating variable method called *over-shooting*. In the exploratory phase, AVM doubles the amount of change with every successful iteration. Eventually it over-shoots, i.e. the fitness value decreases due to an excessive change. The excessiveness is exaggerated in this case because the input values are squared in the predicate that determines the formation of a right-angle triangle. Once it over-shoots, AVM switches to pattern phase and starts to change the next input variable. If the desired solution is sufficiently hard to find, there is a chance that AVM will keep over-shooting with oscillating fitness values. IG fails to achieve full branch coverage for these predicates because it spends the allowed maximum fitness evaluations while over-shooting.

Figure 5.6-5.8 and Figure 5.10 show that the reaction of `complexbranch` to changes in interaction level and search radius is different from that of other subject programs. The number of fitness evaluations does not peak at the interaction level we predicted. Increasing interaction level does not seem to have a positive impact on either coverage or mutation score. Finally, increasing search radius does seem to have a negative impact on mutation score. Unlike `triangle1` and `triangle2`, we could not identify a clear reason why it behaves so differently. It does not contain the type of data dependency observed in `triangle1`; the only notable difference between `complexbranch` and other subject programs was the fact that `complexbranch` uses a `switch` statement, which is essentially equivalent to nested `if` statements and should not present any particular challenge. The differences may be caused by the fact that `complexbranch` has the largest

input domain among the subject programs, with complex branch predicates that are specifically designed to make it hard to achieve full branch coverage.

5.8 Case Study

This section applies the test data augmentation technique to unit level testing of real world examples that use more complex input data types.

5.8.1 Subject Programs

The case study considers two Java methods from real world Java libraries. The first library is `colt`, which is an open source Java library for High Performance Scientific and Technical Computing developed at European Organisation for Nuclear Research (CERN). From the library, we choose `int binarySearch(int[] list, int key, int from, int to)`, which is an implementation of binary search algorithm. It searches the integer array `list` for the integer `key`, starting from index `from` to `to`. The method contains 5 branches. The second library is SIENA (Scalable Internet Event Notification Architecture), which is an open source event notification framework. From the Java API implementation, we choose `int read_number(byte[] buffer)`, which parses a number from the string contained in byte array `buffer`, and determines the type of the number, i.e. integer, double, or unknown. The method contains 22 branches including one unreachable branch. Since both methods take an array as input, the size of the search space for potential input is unbounded.

5.8.2 Test Data Augmentation Technique

Test input for a dynamic data structure requires more sophisticated modification operators than those used in Section 5.4.4. For `binarySearch`, we consider the following modification opeartors. Given a test input of <`int[] list`, `int key`, `int from`, `int`

`to`>:

- **Insert into list**: inserts a random integer into the integer array. After insertion, the array is sorted in order to meet the precondition of binary search.
- **Replace key**: replaces the value of `key` variable with a randomly chosen number from `list`
- **Increase/decrease from**: increases or decreases the value of `from` variable by 1
- **Increase/decrease to**: increases or decreases the value of `to` variable by 1

For `read_number`, we consider the following modification operators. Given a test input of <`byte[] buffer`>:

- **Insert into buffer**: inserts a random ASCII character at a random position
- **Delete from buffer** : selects a random position in `buffer` and removes the value at the position, which returns a shorter array
- **Increase/decrease byte**: selects a random byte in `buffer` and increases or decreases the stored value by 1

In order to apply any sophisticated set of test data generation tools to arbitrary real world examples, it is not uncommon to find that some small modifications are required relating to implementation details. The case studies upon which we report here are no exception. Specifically, two minor modifications are required for the test data augmentation technique described in Section 5.4.4 in order to cater for the dynamic data structure. First, Euclidean distance needs to be replaced with more generic distance metric. There is some existing work on the generic distance measure between test inputs, but it is still in its early stage and can be computationally expensive [57]. Instead of measuring the distance between two test inputs, we consider the distance between two minimal string representations of the test data and turn to weighted Levenshtein distance [125].

The basic Levenshtein distance is used to measure the distance between two arbitrary string. It is computed as the minimum number of operations needed to transform one string into the other. The operations are insertion, removal, and replacement of a single character. Algorithm 8 describes the dynamic programming approach to compute Levenshtein distance.

Algorithm 8: Pseudo-code of basic Levenshtein Distance calculation algorithm

Input: Two strings, s[1..m] and t[1..n]
Output: Distance d in integer

```
(1)   declare int d[0..m, 0..n]
(2)   for i = 0 to m
(3)       d[i, 0] := i
(4)       for j = 0 to n
(5)           d[0, j] := j
(6)   for i = 1 to m
(7)       for j = 1 to n
(8)           if s[i - 1] = t[j - 1]
(9)               cost := 0
(10)          else
(11)              cost := 1
(12)          d[i, j] := min(d[i−1, j]+1, d[i, j−1]+1, d[i−1, j−1]+cost)
(13)
(14)  return d[m, n]
```

We extend the basic definition of Levenshtein distance to obtain a weighted version, by replacing `cost:=1` in line 11 of Algorithm 8 with `cost:=|s[i-1]-t[j-1]|`, i.e. the difference between byte representation of two characters at index `i-1`. This enables us to rank the results of replacement operator according to the number of incremental/decremental changes required for the replacement. For example, the distance between “1,3” and “1,4” is shorter than the distance between “1,3” and “1,7”.

The second modification actually simplifies the algorithm described in Algorithm 5. The modification operators used in the empirical study are deterministic, and so is Algorithm 6, the algorithm that generates the neighbourhood solutions. However, some of the new modification operators introduced in this section are inherently random,

which means the test data augmentation algorithm is likely to consider different set of neighbourhood solutions with each restart of the hill climbing. Therefore, the algorithm used in the case study adopts the steepest ascent hill climbing.

5.8.3 Iguana : Hill Climbing Test Data Generation

The results of the test data augmentation technique is compared to the Hill Climbing algorithm based test data generating using Iguana tool. The specifications of the subject programs require that special care is taken to use Iguana to generate test data for the programs. For example, `binarySearch` takes <`int[] list`, `int key`, `int from`, `int to`> as input. It is implied in the specification that variables `from` and `to` should contain integers that can be index values of the array `int[] list`. However, this is not reflected in the form of branch distance, preventing the Hill Climbing algorithm from receiving any guidance for the values of `from` and `to`. Therefore, we have limited the length of a list to 10 and adjusted the input domain for variable `from` and `to` to $[0 \ldots 9]$. The variable `key`, on the other hand, is used in a branch that determines whether the binary search has found the `key` variable in the `list`. This enables the Hill Climbing algorithm to receive guidance for the value of `key`. Finally, the contents of the integer array is expected to be in a sorted order by the specification of the binary search. However, there is no way to force Iguana to form a sorted array and the requirement is ignored as a result. This leads to an interesting side-effect, which is discussed in Section 5.8.6.

In case of `read_number`, the only constraint implied by the input specification is that the byte array contains characters. Therefore, we have limited the input domain for each character in the byte array to that of ASCII code, $[0 \ldots 255]$.

5.8.4 Original Test Suite and Mutation Faults

The original test suites were generated manually so that there exists one test case per each branch in the program, which is covered by the test case. Therefore, the test suite for

Subject	$\bar{n}_{IG}$	$\sigma^2_{n_{IG}}$	$\bar{n}_{TA}$	$\sigma^2_{n_{TA}}$	p-value
`binarySearch`	475.70	5136.85	556.20	940.84	1.0
`read_number`	23,370.30	14,631,258.00	574.00	1,367.37	$< 10^{-15}$

Table 5.6: Average number of fitness evaluations ($\bar{n}$) and variance (σ^2_n) for TA and IG.

Subject	c_{OR}	$\bar{c}_{IG}$	$\sigma^2_{c_{IG}}$	$\bar{c}_{TA}$	$\sigma^2_{c_{TA}}$	p-value
`binarySearch`	100.00%	100.00%	0.00	100.00%	0.00	n/a
`read_number`	95.54%	76.36%	75.25	95.23%	1.03	$< 10^{-8}$

Table 5.7: Average branch coverage ($\bar{c}$) and variance (σ^2_c) for TA and IG.

`binarySearch` contains 5 test cases, which collectively achieve 100% branch coverage. The test suite for `read_number` contains 21 test cases (22 branches, 1 unreachable). Due to an unreachable branch in the code, the maximum branch coverage that can be achieved from `read_number` is 95.4%.

Mutation faults are generated by `muJava` mutation tool using the same set of mutation operators shown in Table 5.1. This resulted in 61 executable mutants in case of `binarySearch` and 190 executable mutants in case of `read_number`.

5.8.5 Evaluations

In order to cater for the potential impact upon the result of the inherent randomness of the search-based test data augmentation, the experiment for each subject is repeated for 20 times. For the case study, the interaction level of test data augmentation algorithm is set to 1, i.e. only one input variable is modified for each neighbouring solution. The search radius is set to 20. As in the empirical study, the efficiency of the technique is measured by counting the number of fitness evaluations, whereas the effectiveness of the technique is measured both by mutation score and by branch coverage. Iguana tool is configured to spend the maximum of 3,000 fitness evaluations for a single branch before moving on to the next branch.

5.8.6 Results and Analysis

Table 5.6 shows the statistical analysis of the average number of fitness evaluations required by IG and TA for the generation of the additional test suite. For `binarySearch`, IG turns out to be more efficient than TA, being contrary to the trend observed in smaller programs. This is due to the fact that Iguana ignores the requirements that the input array is sorted. When an unsorted array is given to a binary search algorithm, it is very difficult for the search to be successful but it remains relatively easy to cover some branches before terminating. For example, the first step of a binary search is to compare the middle element of the array to the key variable to determine which one is bigger.

A random assignment of the array and the key variable will have an average of 50% chance of satisfying either branch of this step, which is an advantage for IG as it assigned the initial values randomly. However, while they succeed in covering branches, very few of the test cases generated by IG will conform to the *normal* behaviour of a binary search. On the other hand, TA starts with a legitimate test suite, i.e. the array elements are sorted. The definition of the fitness function in Section 5.4.3 requires that, when generating an additional test case, TA should preserve the execution trace of the original test case; that is, TA can only produce test cases that conform to the normal behaviour of a binary search, because the manually generated original test suite does. This limits the scope of acceptable modifications to the original test case, resulting in suboptimal performance.

By contrast, the results for `read_number` follow the trend observed in smaller programs. This is because `read_number` shares the similar type of data dependency we observed in `triangle1`. A few branches in `read_number` contain predicates that depend on either a variable that was assigned earlier or a return value of an external function. For the same reason we discussed in Section 5.7, IG is vulnerable to this type of data dependency in branch predicates. As a result, TA is more efficient than IG in case of `read_number`.

The average number of fitness evaluations required by IG and TA are compared using one-sided t-test with the significance level of 95%. The null hypothesis is that there is no difference between mean values of n_{IG} and n_{TA}. The alternative hypothesis is that $n_{IG} > n_{TA}$. For `binarySearch`, the alternative hypothesis is rejected for the reason described above. For `read_number`, the alternative hypothesis is accepted.

Table 5.7 shows the average branch coverage achieved by the original test suite, IG and TA respectively. For `binarySearch`, both technique reproduce the 100% branch coverage of the original test suite with variance of 0. In `read_number`, there exists a single branch that is unreachable, making the highest achievable branch coverage 95.54%. The average branch coverage achieved by TA almost reaches the highest possible value; in fact, TA successfully reaches 95.54% branch coverage 18 times out of the 20 runs, as can be observed in the relatively small variance. On the other hand, the average branch coverage achieved by IG is 76.36%, with higher variance than TA.

The average branch coverage achieved by IG and TA are compared using one-sided t-test with the significance level of 95%. The null hypothesis is that there is no difference between mean values of c_{IG} and c_{TA}. The alternative hypothesis is that $c_{TA} > c_{IG}$. The results from `binarySearch` does not require the t-test as both samples are essentially uniform and equal to each other. For `read_number`, the alternative hypothesis is accepted at 95% significance level.

Figure 5.11 shows the average mutation score achieved by IG and TA, compared to the original mutation score. Plots for IG and TA show the average mutation score from 20 runs. While both IG and TA achieve a higher mutation score than the original test suite, TA shows the highest mutation score in both programs. Table 5.8 shows the statistical analysis of the mutation score results in detail. The mutation scores of IG and TA are compared using one-sided t-test. The null hypothesis is that there is no difference between mean values of m_{IG} and m_{TA}. The alternative hypothesis is that $m_{IG} < m_{TA}$. While $\bar{m}_{IG}$ is less than $\bar{m}_{IG}$ for `binarySearch`, the observed p-value narrowly rejects

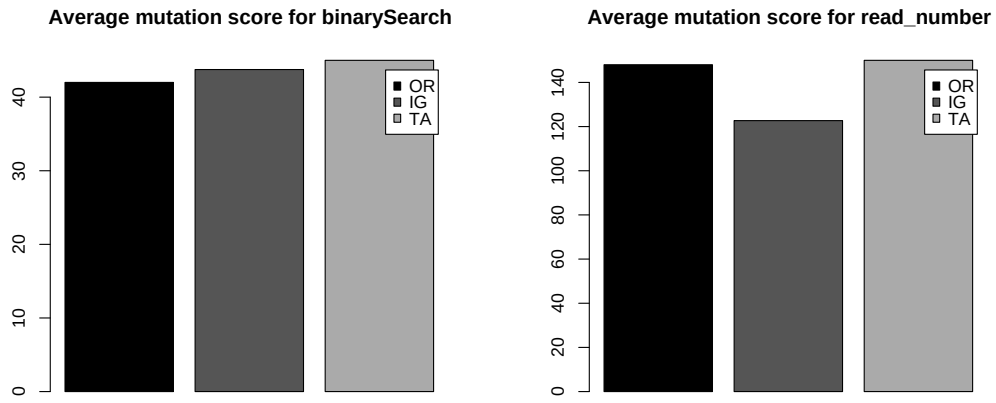

Figure 5.11: Average mutation score for OR, TA and IG. For both `binarySearch` and `read_number`, TA scores the highest mutation score on average.

Subject	m_{OR}	$\bar{m}_{IG}$	$\sigma^2_{m_{IG}}$	$\bar{m}_{TA}$	$\sigma^2_{m_{TA}}$	p-value
`binarySearch`	42.00	43.65	7.78	45.00	6.63	0.07
`read_number`	148.00	122.70	418.64	150.00	5.68	$< 10^{-5}$

Table 5.8: Average number of fitness evaluations ($\bar{n}$) and variance (σ^2_n) for TA and IG.

the alternative hypothesis, i.e. there is no statistically significant difference between $\bar{m}_{IG}$ and $\bar{m}_{IG}$. The observed p-value for `read_number` confirms the alternative hypothesis, i.e., TA achieves a higher mutation score than IG with statistical significance.

In order to make a detailed comparison between the original test suite and the test suites generated by IG and TA, we classify the mutation faults according to the test suite that killed it. Figure 5.12 shows the resulting Venn diagrams for the results of mutation testing. If, during the 20 executions, a mutation fault is killed by a test suite at least once, the fault is classified as being killed by the test suite. The diagram for `binarySearch` shows that, although there is no statistically significant difference between mutation scores of IG and TA, two techniques kill different set of mutation faults. The

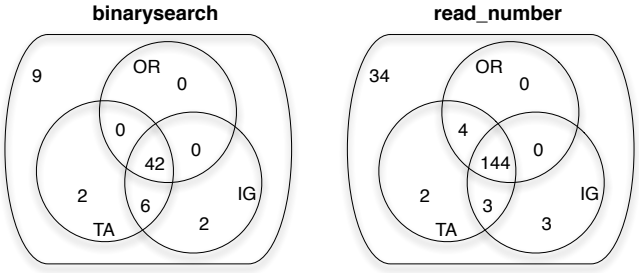

Figure 5.12: Venn diagrams classifying mutation faults according to the test suites that killed them.

diagram for `read_number` also shows that TA is capable of detecting a set of mutation faults that are not detected by other techniques. Finally, it can be observed that, as with the empirical studies in Section 5.6, both $|TA - OR|$ and $|IG - OR|$ are greater than 0, confirming the added value of executing additional test cases.

5.9 Threats to Validity

Threats to internal validity concern the factors that might have affected the comparison of IG and TA. The two techniques are so inherently different from each other that the comparison at algorithmic level may not be sound. However, should a tester want a new and additional test suite, the only available state-of-the-art solution is to adopt existing test data generation technique and hope that it will generate a completely new test suite. One alternative is to modify existing search-based test data generation technique to be existing test suite aware. The modified fitness function will measure not only branch distance based fitness, but also the distance from the existing test data. However, this can only add more complexity to traditional search-based test data generation techniques.

Both techniques are inherently stochastic, which may affect the comparison. The experiments are repeated for 20 times to cater for the stochastic nature, and the results are verified with statistical testing. For some experiments, the observed measurements

are consistent and show little variance.

With TA, the quality of the original test suite can affect the quality of the new test suite generated by TA. The original test suites are manually generated for each input domain with the sole purpose of achieving full branch coverage. That is, other testing concerns such as boundary values or mutation score are not considered. However, there still exists a chance that the original test suites are biased.

As described in Section 3.5.1, Welch's t-test assumes a parametric distribution of the samples. While the exact distribution of the sample is not known in this empirical study, it will approximate a normal distribution with a sufficiently large sample size according to the central limit theorem [161].

Threats to external validity concerns the factors that limit generalisation of the results. One issue is the representativeness of the subject programs, which are relatively small-scale examples. It cannot be guaranteed that the observed results of this study will extrapolate to larger-scale real world programs. This can be addressed only by further study with larger programs. However, the subject programs have been utilised as benchmarks for test data generation techniques [140, 195, 211]. They also provide non-trivial search space in combination with different input domains. Another issue concerns the selection of the test data generation technique for the comparison. This chapter compares the proposed search-based test data augmentation technique to a hill climbing based AVM (Alternating Variable Method). Other test data generation techniques may produce different results. However, AVM has been known to be as much as, or even better than other search techniques such as genetic algorithms, for certain classes of test data generation problems [73].

Threats to construct validity arise when the measurements in the experiments do not capture the concepts that they are supposed to represent. The efficiency measure is a count of fitness evaluations, which equals the number of times the program under test is executed. It does not consider other costs of testing such as generating and checking test

oracles. However, as test data augmentation is applied to larger programs, the execution time of the program under test is at least one of the major elements that account for the cost of testing for any type of dynamic testing. Similarly, the effectiveness measure used in this chapter may not capture the real effectiveness of a test suite; its fault detection capability. However, structural coverage and mutation score has been widely used as successful surrogate of fault detection capability in software testing literature [45, 140, 149, 195, 211]. Also, the testing process can benefit from the existence of an alternative test suite even when it does not detect any additional faults. For example, the estimation of program reliability when no fault is detected can benefit from alternative test suites.

5.10 Conclusions

This chapter proposed a novel approach to regression testing called *on-demand* regression testing. In on-demand regression testing, the tester generates, in each regression cycle, a novel set of test data that still satisfy the testing goal, using the existing test data as a starting point. This approach ensures that the latency of the regression testing process remains high, while preventing the test suite from growing too large. The key to this approach is a low cost method of generating new test data from existing data while still maintaining their capability of satisfying the testing goals.

The chapter introduced and evaluated a search-based test data augmentation technique, which is a novel method of generating test data from existing test data. Test data augmentation is based on two observations. First, it is beneficial to generate additional test data even if test data are already available. Second, if there are existing test data, generating additional test data can be much more efficient when it utilises the knowledge of existing test data.

This chapter introduces a search-based test data augmentation algorithm and instantiates it with structural coverage testing criteria. The proposed algorithm is empirically

compared in terms of its efficiency and effectiveness to a state-of-art test data generation technique. The results show that there exists an attractive trade-off between efficiency and effectiveness of test data augmentation. The cost can be reduced by two orders of magnitude for some cases, while achieving competitive structural coverage and mutation score. Test data augmentation is less affected by the size of input domain compared to existing test data generation techniques that suffer with significantly large input domain.

Chapter 6

Incorporating Expert Knowledge To Test Case Prioritisation

6.1 Introduction

Test case prioritisation seeks to find an efficient ordering of test case execution for regression testing. The ideal ordering of test case execution is one that reveals faults earliest. Since the nature and location of actual faults are generally not known in advance, test case prioritisation techniques have to rely on available surrogates for prioritisation criteria. Structural coverage, requirement priority and mutation score have all previously been utilised as criteria for test case prioritisation [43, 53, 193]. However, there is no single prioritisation criterion whose results dominate the others.

One potentially powerful way to enhance a prioritisation criterion is to utilise domain expert judgement by asking the human tester to compare the importance of different test cases. A competent human tester can provide rich domain knowledge about the System Under Test (SUT), including knowledge about logical effects of recent changes and rationale behind the existing test cases. Human guidance may also be required in order to take account of the many implicit, unstated reasons that the tester may have

for favouring one test case over another. If this human guidance is not accounted for, the tester may reject the proposed order suggested by a prioritisation algorithm.

Prioritisation involving human judgement is not new. The Operations Research community has developed techniques including the Analytic Hierarchy Process (AHP) algorithm [184] that help decision makers to prioritise tasks. However, prioritisation techniques that involve humans present scalability challenges. A human tester can provide consistent and meaningful answers to only a limited number of questions, before fatigue starts to degrade performance. Previous empirical studies show that the largest number of pair-wise comparisons a human can make consistently is approximately 100 [7]. Unfortunately, useful test suites often contain many test cases, potentially requiring considerably more than 100 comparisons.

To address this problem, this study uses clustering algorithms to reduce the cost of human-interactive prioritisation. In our approach, the human tester prioritises, not the individual test cases, but clusters of 'similar' test cases. With a very simple clustering technique, such as agglomerative hierarchical clustering [86], it is possible to generate an arbitrary number of clusters. This allows for control of the number of comparisons presented to the human tester. The reduced number of required comparisons makes it feasible to apply expert-guided prioritisation techniques to much larger data sets. The chapter presents results on the scalability potential of this clustering approach.

The AHP-based prioritisation technique is empirically compared to coverage-based prioritisation techniques using the APFD (Average Percentage of Fault Detection) metric. In order to model various possible human behaviours, we introduce an error model. This allows us to empirically explore the robustness of our approach in the presence of varying degrees of human 'bias' (giving guidance that draws the algorithm away from fault finding test cases). The results show that AHP-based prioritisation is robust; it can outperform coverage-based prioritisation even when the human tester provides misleading answers to comparison questions.

The rest of the chapter[1] is organised as follows. Section 6.2 introduces the cluster-based prioritisation technique used in the study. Section 6.3 describes the Analytic Hierarchy Process and the user model that is used by the empirical evaluation. Section 6.4 explains the details of the empirical study, the results of which are presented in Section 6.5. Section 6.6 presents related work and Section 6.7 concludes.

6.2 Clustering Based Prioritisation

6.2.1 Motivation

A pair-wise comparison approach for prioritisation requires $O(n^2)$ comparisons. While redundancy may make pair-wise comparison very robust, the high cost has prevented it from being applied to test case prioritisation. For example, AHP has been well studied in the Requirements Engineering field. The maximum number of comparisons a human can make consistently is approximately 100 [7]; above this threshold, inconsistency grows significantly, leading to reduced effectiveness.

In order to require less than 100 pair-wise comparisons, the test suite could contain no more than 14 test cases. Considering the scale of real world testing projects, the scalability issue presents a significant challenge. For example, suppose there are 1,000 test cases to prioritise; the total number of required pair-wise comparisons would be 499,500. It is clearly unrealistic to expect a human tester to provide reliable responses for such a large number of comparisons.

This study aims to reduce the number of comparisons required for the pair-wise comparison approach through the use of clustering. Instead of prioritising individual test cases, clusters of test cases are prioritised using techniques such as AHP. From the prioritised clusters, the ordering between individual test cases is then generated.

[1]This chapter is an extended version of the author's ISSTA paper: S. Yoo, M. Harman, P. Tonella and A. Susi, Clustering Test Cases To Achieve Effective & Scalable Prioritisation Incorporating Expert Knowledge. *Proceedings of the 2009 International Symposium on Software Testing and Analysis (ISSTA)*, Chicago, USA, *pages 201-211.*

6.2.2 Clustering Criterion

The clustering process partitions objects into different subsets so that objects in each group share common properties. The clustering criterion determines which properties are used to measure the commonality. When considering test case prioritisation, the ideal clustering criterion would be the similarity between the faults detected by each test case. However, this information is inherently unavailable before the testing task is finished. Therefore, it is necessary to find a surrogate for this, in the same way as existing coverage-based prioritisation techniques turn to surrogates for fault-detection capabilities.

In this study we utilise dynamic execution traces of each test case as a surrogate for the similarity between features tested. Execution of each test case is represented by a binary string. Each bit corresponds to a statement in the source code. If the statement has been executed by the test case, the digit is 1; otherwise it is 0. The similarity between two test cases is measured by the distance between two binary strings using Hamming distance.

Algorithm 9: Agglomerative Hierarchical Clustering

Input: A set of n test cases, T
Output: A dendrogram, D, representing the clusters
(1) Form n clusters, each with one test case
(2) $C \leftarrow \{\}$
(3) Add clusters to C
(4) Insert n clusters as leaf node into D
(5) **while** there is more than one cluster
(6) Find a pair of clusters with minimum distance
(7) Merge the pair into a new cluster, c_{new}
(8) Remove the pair of test cases from C
(9) Add c_{new} to C
(10) Insert c_{new} as a parent node of the pair into D
(11) **return** D

6.2.3 Clustering Method

We use a simple agglomerative hierarchical clustering technique. Its pseudo-code is described in Algorithm 9 below:

The resulting dendrogram is a tree structure that represents the arrangement of clusters. Figure 6.1 shows an example dendrogram. It is possible to generate k clusters for any k in [1, n] by cutting the tree at different heights.

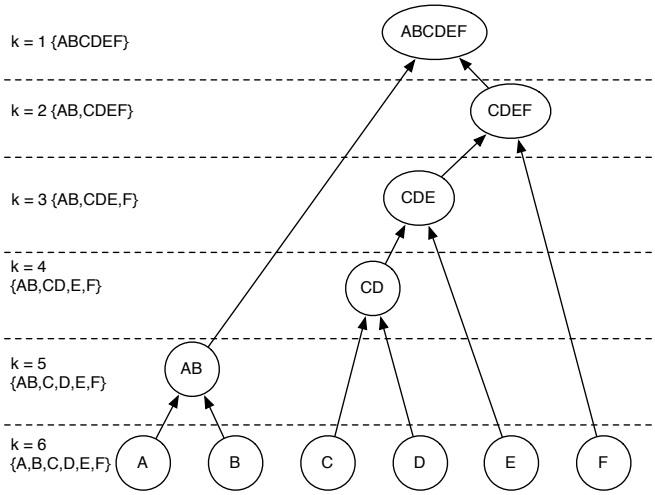

Figure 6.1: An example dendrogram from agglomerative hierarchical clustering. Cutting the tree at different height produces different number of clusters.

6.2.4 Interleaved Clusters Prioritisation

Prioritisation of a clustered test suite is a different problem from the traditional test case prioritisation problem. Two separate layers of prioritisation are required in order to prioritise a clustered test suite. *Intra-cluster* prioritisation is prioritisation of test cases that belong to the same cluster, whereas *inter-cluster* prioritisation is prioritisation of clusters. This chapter introduces the Interleaved Clusters Prioritisation (ICP) process that uses both layers of prioritisation.

It would be more advantageous to *interleave* clusters of test cases than to execute

an entire cluster before executing the next. The latter approach would result in repeatedly executing similar parts of SUT before the prioritisation technique chooses the next cluster; if these test cases reveal similar sets of faults, the rate of fault detection would be less than ideal because the prioritisation technique will reveal a similar set of faults repeatedly. The former approach will avoid this by switching clusters whenever it chooses the next test case.

In ICP, intra-cluster prioritisation is performed first. Based on the results of intra-cluster prioritisation, each cluster is assigned a test case that represents the cluster. Using these representatives, ICP performs inter-cluster prioritisation. The final step is to interleave prioritised clusters using the results of both intra- and inter-cluster prioritisation.

More formally, suppose a test suite TS is clustered into k clusters, $C_1, \ldots, C_k$. After intra-cluster prioritisation, we obtain ordered sets of test cases, $OC_1, \ldots, OC_k$. Let $OC_i(j)$ be the jth test case in cluster OC_i. Each ordered set OC_i is then represented by $OC_i(1)$ in the inter-cluster prioritisation, which will produce OOC, an ordered set of $OC_i(1 \leq i \leq k)$. Let OOC_i be the ith cluster in OOC. The interleaving process is described in pseudo-code in Algorithm 10.

Algorithm 10: Interleaved Clusters Prioritisation

Input: An ordered set of k ordered clusters, OOC
Output: An ordered set of test cases, OTC

```
(1)  OTC =<>
(2)  i ← 1
(3)  while OOC is not empty
(4)      Append OOC_i(1) to OTC
(5)      Remove OOC_i(1) from OOC_i
(6)      if OOC_i is empty then Remove OOC_i from OOC
(7)      i ← (i + 1) mod k
(8)  return OOC
```

For example, given $OOC =<< t_3, t_1 >, < t_4, t_2 >, < t_5 >>$, the result of Algorithm 10 will be a sequence of test cases, $< t_3, t_4, t_5, t_1, t_2 >$. Note that ICP does not

presume any specific choice of prioritisation technique. Any existing test case prioritisation technique can be used for either intra-cluster or inter-cluster prioritisation.

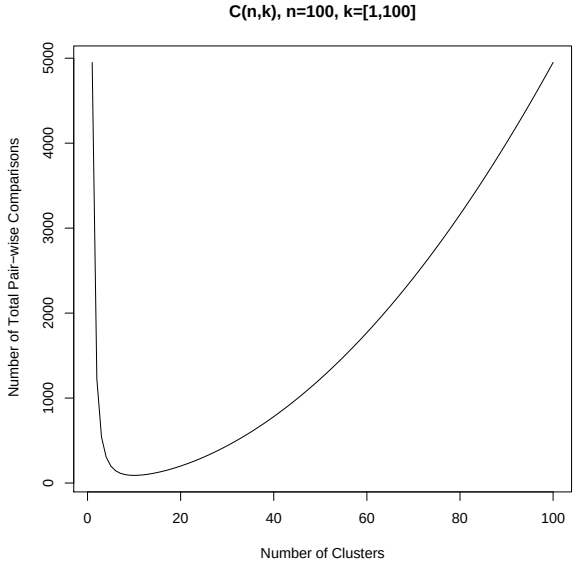

Figure 6.2: Plot of average number of pair-wise comparisons required for k cluster-based prioritisation of 100 test cases.

6.2.5 Cost of Pair-wise Comparisons

Since pair-wise comparisons require human intervention, the cost of any pair-wise comparison approach largely depends on the number of comparisons required. When the pair-wise comparison approach is used both for intra- and inter-cluster prioritisation, the number of comparisons required for ICP is the sum of the cost of intra-cluster prioritisation and inter-cluster prioritisation. Given a test suite of size n clustered into k clusters, each cluster contains $\frac{n}{k}$ test cases on average. The average number of comparisons for intra-cluster prioritisation is $k \cdot \frac{1}{2}\frac{n}{k}(\frac{n}{k} - 1)$. The number of comparisons of inter-cluster prioritisation is computed simply as $\frac{k(k-1)}{2}$. Therefore, the average total cost of pair-wise ICP for a test suite of size n and k clusters, $C(n, k)$, is $\frac{k(k-1)}{2} + k \cdot \frac{\frac{n}{k}(\frac{n}{k}-1)}{2}$.

For all positive n, there exists a specific value of k that minimises $C(n, k)$. Figure 6.2

illustrates $C(n, k)$ when $n = 100$ and $1 \leq k \leq n$. The maximum cost, with no clustering, is 4,950 comparisons. With clustering, the minimum cost is 381 when $k = 17$. While the reduction is by an order of magnitude, the minimum cost of 381 is still too expensive for a human tester to consider.

To further reduce the cost, ICP used in the chapter is hybridised so that intra-cluster prioritisation uses the traditional coverage-based greedy prioritisation algorithm. The human tester is only involved with inter-cluster prioritisation. The cost of hybrid approach is, therefore, only the number of comparisons required for inter-cluster prioritisation, which is $C(n, k) = \frac{k(k-1)}{2}$. To ensure that fewer than 100 comparisons are required, we use hybrid-ICP with $k = 14$ throughout the study, which results in 91 comparisons.

6.2.6 Suitability Test

ICP is most effective when the result of clustering is semantically significant, i.e. test cases that execute similar parts of SUT belong to the same cluster. As k decreases, the semantic significance of clustering is also diminished, since eventually the clustering algorithm will start to place semantically different test cases in the same cluster. Therefore, hybrid ICP with $k = 14$ may not work well with every test suite/SUT combination.

Since any form of human involvement in test case prioritisation is a significant commitment, applying hybrid ICP to a combination of test suite and SUT that is not suitable would be a waste of resources. A decision is required as to whether it is worth applying the hybrid ICP. To support this decision making process, we propose an automated suitability test that does not require human judgement. The test is an automated ICP, fully based on structural coverage. Both intra- and inter-cluster prioritisation is performed based on structural coverage. It also uses fault detection information using faults that belong to the AR (Already Revealed) fault set. If the result of the test is not worse than traditional coverage-based prioritisation techniques, it would confirm that clustering is not detrimental to the performance of ICP, in which case replacing inter-cluster

prioritisation with the pair-wise comparison approach is likely to have a positive impact on the rate of fault detection with the unknown faults that belong to the TBR (To Be Revealed) fault set.

6.3 Analytic Hierarchy Process

6.3.1 Definition

In order to prioritise n items, AHP requires all possible pair-wise comparisons between n items. Comparisons are represented using the scale of preference described in Table 6.1.

p_{ij}	Preference
1	i is equally preferable to j
3	i is slightly preferable over j
5	i is strongly preferable over j
7	i is very strongly preferable over j
9	i is extremely preferable over j

Table 6.1: Scale of preference used in the comparison matrix of AHP

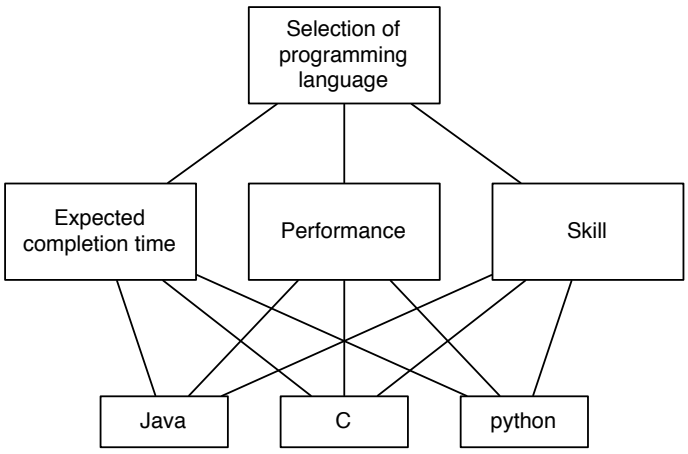

Figure 6.3: An example hierarchy between comparison criteria for AHP

Note that the preference relation is not necessarily transitive. The decision maker is

entitled to give answers such as $A \succ B, B \succ C$ and $C \succ A$. In other words, p_{ik} is not necessarily equal to $p_{ij} \cdot p_{jk}$. This lack of transitivity in the preference relation allows AHP to cope with inconsistencies given by the decision maker. However, these inconsistencies are mitigated by the high redundancy available due to multiple comparisons. By definition, the scale is a ratio-based measurement. That is, given p_{ij}, p_{ji} is defined as $\frac{1}{p_{ij}}$.

The result of comparisons are combined in an n by n matrix, M. Naturally, $M(i,i) = 1(1 \leq i \leq n)$. For the rest:

$$\forall i(1 \leq i \leq n) \forall j(1 \leq j \leq n \wedge i \neq j), M(i,j) = p_{ij}$$

The priority weighting vector E is the eigenvector of a matrix M', which is calculated from M by normalising over the columns. E is calculated by taking the average across rows of M':

$$M'(i,j) = \frac{M(i,j)}{\sum_{1 \leq k \leq n} M(i,k)}$$

$$E_i = \frac{\sum_{1 \leq k \leq n} M(k,i)}{n}$$

It is also possible to construct a hierarchy of multiple criteria [184]. To illustrate how this hierarchical AHP is achieved, suppose that AHP is used to select the programming language that will be used to implement a system. Management cares about expected completion time, performance of implemented system and programmers' skill in each language. If the candidate languages are {Java, C, Python}, then the hierarchy of criteria is as in Figure 6.3. First, programming languages are evaluated against each criterion in the middle level. This produces a set of priority weighting vectors, V. Second, the criteria in the middle level are prioritised, i.e., the relative importance between completion time, performance and skill level is determined using AHP. This produces another priority

weighting vector, E. The final weighting vector is calculated by calculating the weighted sum of the vectors in V; for each criterion, the weight is given by E.

6.3.2 User Model

The approach proposed in this study will be evaluated with respect to an 'ideal user model' and more 'realistic user' model that simulates human errors. The *ideal* user provides comparisons that are consistent with the real fault detection capability of test cases. While this may be over-optimistic, it provides an upper limit on the effectiveness of cluster-based test case prioritisation using AHP.

Since AHP allows the user to compare two entities with degrees of preference rather than simple binary relations, the ideal user model needs to consider how to quantitatively differentiate the relative importance of test cases. Previous work using human input for test case prioritisation only required binary relations, which are obtained by checking which test case detects more faults than the other. We derive varying degrees of relative importance by checking how much difference there is between the number of faults detected by two test cases.

Suppose two test cases t_A and t_B are being compared. Let n_A be the number of faults detected by t_A, and n_B by t_B. The 'ideal' user model used in the study sets the scale of preference between t_A and t_B, p_{AB}, as shown in Table 6.2.

Condition	p_{AB}	Description
$n_A = n_B$	1	Equal
$n_A > 0$ and $n_B = 0$	7	Very Strongly prefer t_A
$n_A > 0, n_B > 0, n_A \geq 3n_B$	9	Extremely prefer t_A
$n_A > 0, n_B > 0, n_A \geq 2n_B$	7	Very Strongly prefer t_A
$n_A > 0, n_B > 0, n_A \geq n_B$	5	Strongly prefer t_A
$p_{BA} = \frac{1}{p_{AB}}$		

Table 6.2: Scale of preference for the 'ideal user' model used in Chapter 6

For a more realistic user model, we introduce a model of human error. Suppose that test case t_A and t_B are being compared with the result of p_{AB}. Let p'_{AB} be the result with error. There are eight types of errors; Table 6.3 shows all eight types of error.

Errors of type 1 and 2 occur when a human tester claims that two test cases are equally important, when in fact one of them is more important than the other. Errors of type 3 and 4 occur when a human tester claims that a test case is more important than the other, when in fact it is the opposite. Errors of type 5 and 6 occur when a human tester claims that a test case is more important than the other when in fact they are equally important. Finally, errors of type 7 and 8 occur when a human tester correctly claims the inequality relation between two test cases, but answers the ratio of relative importance incorrectly. In order to only include errors that mean the human judgement is definitely wrong, only errors of type 1 to 6 are considered in the empirical studies.

Type	Original	Error
1	$p_{AB} > 1$	$p'_{AB} = 1$
2	$p_{AB} < 1$	$p'_{AB} = 1$
3	$p_{AB} > 1$	$p'_{AB} < 1$
4	$p_{AB} < 1$	$p'_{AB} > 1$
5	$p_{AB} = 1$	$p'_{AB} > 1$
6	$p_{AB} = 1$	$p'_{AB} < 1$
7	$p_{AB} > 1$	$p'_{AB} > 1$ and $p'_{AB} \neq p_{AB}$
8	$p_{AB} < 1$	$p'_{AB} < 1$ and $p'_{AB} \neq p_{AB}$

Table 6.3: Classification of different types of errors that the human tester can make for AHP

In order to avoid any bias, we use the simplest uniformly distributed error model. Given an error rate e, which is a real number between 0 and 1, the model provides a correct answer with probability $1 - e$. With probability e, an error is introduced by changing the result of the comparison to one of alternative results defined by the types of errors with a uniformly distributed probability.

6.3.3 Hierarchy

AHP can deal with multiple prioritisation criteria if the user can specify relative importance between different criteria. In the proposed prioritisation technique, both a single criterion hierarchy model and multiple criteria hierarchy model are utilised. The single criterion hierarchy model requires only the comparisons from the human expert. This model allows us to observe how well the proposed technique performs when expert knowledge is used alone. However, the model may produce sub-optimal prioritisation because comparisons only reveal information about the relative quantity of faults found by test cases, not their location. For example, consider the test cases shown in Table 6.4. The ideal human expert will make pair-wise comparisons $t_1 \prec t_2$, $t_1 \prec t_3$, and $t_2 \prec t_3$. Since these comparisons are consistent with each other, the sequence provided by AHP is $< t_1, t_2, t_3 >$. However, the optimal ordering is $< t_1, t_3, t_2 >$ because of the overlap in detected faults between t_1 and t_2.

Test Case	Branch 1 (Fault 1)	Branch 2 (Fault 2)	Branch 3 (Fault 3)	Branch 4 (Fault 4)
t_1	x	x	x	
t_2	x	x		
t_3				x

Table 6.4: The optimal sequence is $< t_1, t_3, t_2 >$. However, perfect pair-wise comparisons will result in $t_1 \prec t_2$, $t_1 \prec t_3$, and $t_2 \prec t_3$, which will produce the sub-optimal sequence $< t_1, t_2, t_3 >$.

To overcome this limitation of AHP, the hierarchical AHP model utilises two prioritisation criteria: expert knowledge obtained from pair-wise comparisons and coverage-based prioritisation results. The user comparisons matrix is obtained using the AHP procedure described above. The structural coverage matrix reflects the result of coverage-based prioritisation. Given a result of statement coverage-based prioritisation, the coverage matrix is filled by Algorithm 11. Note that if t_i comes before t_j in statement

coverage-based prioritisation, $p_{t_i t_j}$ is set to preference scale of 3, which is weaker than the values used by the user model as shown in Table 6.2. This is to ensure that the comparisons based on statement coverage should not override the comparisons from the human expert (i.e coverage-based comparisons are made in weaker terms than comparisons by the human expert).

Algorithm 11: Coverage Matrix Generator

Input: A set of n test cases, $T = \{t_1, \ldots, t_n\}$, and an ordered set of prioritised positions of each test case in T, $O =< i_1, \ldots, i_n >$, such that $i_j = k$ means jth test case is t_k
Output: An n by n coverage matrix, M

(1) **for** $i = 1$ **to** $i \leq n$
(2) $M[i][i] = 1.0$ //equal
(3) **for** $j = 1$ **to** $O_i - 1$
(4) $M[O_i][j] = 3$ // slightly favour t_i
(5) **for** $j = O_i + 1$ **to** $n - 1$
(6) $M[O_i][j] = \frac{1}{3}$ //slightly favour t_j

When using multiple criteria, AHP requires the human user to determine the relative importance not only between entities that are being prioritised (i.e. test cases) but also between criteria themselves (i.e. expert knowledge and statement-based prioritisation). Using Table 6.1, this study applies a set of 9 different human-to-coverage preference values, $p_{[H][C]}$, $\{9, \ldots, 1, \ldots, \frac{1}{9}\}$. The ICP with single criterion AHP model will be denoted by ICP_S; ICP with hierarchical AHP model will be denoted by ICP_M.

6.4 Experimental Set-up

6.4.1 Subjects

Table 6.5 shows the subject programs studied in this chapter. They range from 412 to 122,169 LOC. Each program and its test suite is taken from Software Infrastructure Repository (SIR) [41].

`printtokens` and `schedule` are part of Siemens suite. SIR contains multiple test

Program	Test Suite	(Avg.) TS Size	LOC
`printtokens`	4	317.00	726
`schedule`	4	225.25	412
`space`	4	158.50	6,199
`gzip`	1	212	5,680
`sed`	1	370	14,427
`vim`	1	975	122,169
`bash`	1	1061	59,846

Table 6.5: Program & test suite sizes of subject programs studied in Chapter 6

suites for these programs; four test suites are chosen randomly. `space` is an Array Description Language (ADL) interpreter that was developed by European Space Agency. From SIR, four test suites are chosen randomly. `gzip`, `sed`, `vim` and `bash` are widely used Unix programs. Only one test suite is available for these programs. Coverage data for the subject programs are generated using `gcov`, a widely used profiling tool for `gcc`.

6.4.2 Suitability Test Configuration

SIR contains versions with injected faults and the mapping from test cases to the faults detected by each test case; for `gzip`, `sed`, `vim` and `bash`, it contains multiple consecutive versions of the source code and corresponding fault detection information. Whenever available, the empirical study for the suitability testing utilises two distinct, consecutive versions of the program. The faults from the first version are used as AR (Already Revealed), whereas the faults from the second version is used as TBR (To Be Revealed). For `printtokens`, `schedule` and `space`, multiple versions of the source code are available but there exists a single set of fault detection information for a single version. For these programs, we randomly divide the known faults into the AR and TBR sets and re-use the structural coverage recorded from the source code of the same version. Table 6.6 shows the size of group AR and TBR for each program respectively.

Program	Size of AR	Size of TBR	Mult. Ver.
`printtokens`	3	4	No
`schedule`	4	5	No
`space`	18	20	No
`gzip`	2	3	Yes
`sed`	6	4	Yes
`vim`	4	3	Yes
`bash`	4	9	Yes

Table 6.6: Subject programs and the size of AR (Already Revealed) and TBR (To Be Revealed) sets

6.4.3 Evaluation

The results of ICP are compared to the optimal ordering, OP, and statement coverage-based ordering, SC. The optimal ordering is obtained by prioritising the test cases based on the fault detection information. It is impossible to know fault detection record in advance, and therefore, the optimal ordering is not available in the real world. Statement coverage-based ordering is obtained by performing additional-statement prioritisation. This method has been widely studied and known to produce reasonable results [53, 130].

In the first set of empirical studies, we evaluate ICP against OP and SC using Average Percentage of Fault Detection (APFD) metric [51]. Let T be the test suite containing n test cases and let F be the set of m faults revealed by T. For ordering T', let TF_i be the order of the first test case that reveals the ith fault. APFD value for T' is calculated as following:

$$APFD = 1 - \frac{TF_1 + \ldots + TF_m}{nm} + \frac{1}{2n}$$

ICP experiments are performed with different values for user error rates ranging from 0.05 (user is wrong 5 times out of 100 on average) to 1.0 (user is always wrong) in steps of 0.05.

6.4.4 Research Questions

This study considers the following research questions:

RQ1. Effectiveness: How much improvement in effectiveness is achieved by ICP compared to the optimal ordering and coverage-based prioritisation?

RQ2. Configuration: When the human input is combined with other prioritisation criteria such as structural coverage, what is the ideal configuration between human input and other criteria?

In order to answer **RQ1** and **RQ2**, we evaluate two different hybrid-ICP instances, one with single hierarchy model AHP and the other with multiple hierarchy model AHP. For **RQ1**, we measure the APFD of the produced ordering and compare it to those of optimal ordering and statement coverage-based prioritisation. For **RQ2**, we execute the hybrid ICP with multiple hierarchy model AHP. The secondary prioritisation criteria is derived from statement coverage-based prioritisation using the algorithm shown in Section 6.3.3. We measure APFD of the test case sequences obtained by using different values of $p_{[H][C]}$ and compare these to those of optimal ordering and statement coverage-based prioritisaiton. **RQ1** and **RQ2** are answered in Section 6.5.1.

The second part of the empirical studies deals with tolerance and suitability. Since the quality of the results produced by the proposed technique depends directly on the quality of the human user's input, it is necessary to study how high the allowable level of error rate can be while producing results that are better than coverage-based techniques.

RQ3. Tolerance: What is the highest human error rate that can be tolerated while yielding performance superior to the coverage-based techniques?

For suitability study, we apply the automated suitability test to subject programs and their test suites using the AR faults and see if the difference between ICP and coverage-based prioritisation is consistent with the result of effectiveness and efficiency studies.

RQ4. Suitability: How accurately does the automated suitability test predict the successful result of ICP?

RQ3 is answered in the second part of the empirical study, by increasing the error rate and observing the statistics of the results. **RQ4** is answered by performing automated suitability test and comparing the results to those of effectiveness and efficiency study. **RQ3** and **RQ4** are answered in Section 6.5.2.

6.5 Results and Analysis

Subject	printtokens				schedule			
Test Suite	1	2	3	4	1	2	3	4
OP	0.995	0.995	0.997	0.995	0.991	0.995	0.993	0.993
ICP_S	0.806	0.974	0.967	0.868	0.824	0.917	0.952	0.913
SC	0.930	0.992	0.972	0.960	0.806	0.865	0.782	0.844
Subject	space				gzip	sed	vim	bash
Test Suite	1	2	3	4				
OP	0.983	0.985	0.985	0.982	0.996	0.997	0.998	0.999
ICP_S	0.948	0.933	0.930	0.927	0.996	0.905	0.903	0.144
SC	0.899	0.863	0.948	0.947	0.980	0.876	0.899	0.210

Table 6.7: APFD values obtained from ICP_S and ideal user model compared to those of the optimal ordering and the statement coverage-based prioritisation. Cells with grey background represent the fact that ICP_S outperformed statement coverage-based prioritisation in terms of APFD.

6.5.1 Effectiveness & Configuration

Table 6.7 shows the APFD values measured from the single-hierarchy model approach, ICP_S. The cells with grey background denote configurations that outperformed statement coverage-based prioritisation in terms of APFD metric. The proposed technique only outperforms coverage-based techniques with the right combination of program and test suite. For example, suite 1 and suite 2 of `space` show improvement over SC, but suite 3 and suite 4 do not. The clustering also has detrimental effect on the test suite of `bash`; the APFD value produced by ICP_S is lower than that of SC. Overall, out of 16 prioritisation problems, ICP_S produces higher APFD than statement coverage-based prioritisation for 9 cases. Note that these results assume the human input from the *ideal* user and, therefore, the results are deterministic. The increases in APFD range from 0.5% to 21.8% with average increase of 6.5%. This provides an answer to **RQ1**.

Table 6.8 shows the results from the multiple-hierarchy model approaches, ICP_M. Each configuration uses different $p_{[H][C]}$ value to prioritise the criteria, i.e., comparisons made by human expert and statement coverage-based prioritisation. One trend observed in every prioritisation is that higher $p_{[H][C]}$ tends to produce higher APFD metric values. With a few exceptions, observed APFD values monotonically decrease as $p_{[H][C]}$ decreases. This implies that the ideal configuration for the hybrid ICP_M approach is to set $p_{[H][C]} = 9$, i.e. to favour the human judgement extremely. This provides an answer to **RQ2**. With $p_{[H][C]} = 9$, the increases in APFD range from 0.7% to 22% with average increase of 6.4%.

6.5.2 Tolerance & Suitability

Now we turn to the second set of research questions. Regarding the tolerance study and **RQ3**, Figure 6.4 and Figure 6.5 show how APFD values from ICP_S and ICP_S deteriorate as error rate increases from 0.05 to 1.0. Comparing Figure 6.4 and Figure 6.5 with Table 6.7 and Table 6.8, test suites for which ICP does not achieve improvement

Subject		printtokens				schedule			
Test Suite		1	2	3	4	1	2	3	4
OP		0.995	0.995	0.997	0.995	0.991	0.995	0.993	0.993
	9	0.807	0.974	0.967	0.871	0.825	0.916	0.954	0.912
	7	0.807	0.974	0.967	0.871	0.825	0.916	0.954	0.912
	5	0.807	0.974	0.967	0.871	0.825	0.915	0.954	0.912
ICP_M	3	0.807	0.974	0.966	0.871	0.825	0.914	0.952	0.912
$p_{[H][C]}$	1	0.807	0.974	0.966	0.871	0.824	0.915	0.951	0.912
	1/3	0.808	0.973	0.966	0.870	0.823	0.905	0.945	0.909
	1/5	0.807	0.973	0.966	0.870	0.820	0.903	0.943	0.907
	1/7	0.807	0.973	0.966	0.870	0.821	0.901	0.941	0.906
	1/9	0.806	0.973	0.966	0.870	0.821	0.901	0.941	0.904
SC		0.930	0.992	0.972	0.960	0.806	0.865	0.782	0.844
Subject		space				gzip	sed	vim	bash
Test Suite		1	2	3	4				
OP		0.983	0.9852	0.985	0.982	0.996	0.997	0.998	0.999
	9	0.946	0.938	0.931	0.927	0.996	0.905	0.905	0.144
	7	0.946	0.939	0.931	0.926	0.996	0.905	0.905	0.144
	5	0.947	0.939	0.931	0.926	0.996	0.905	0.905	0.144
ICP_M	3	0.946	0.940	0.931	0.926	0.996	0.905	0.905	0.144
$p_{[H][C]}$	1	0.946	0.939	0.930	0.923	0.996	0.905	0.904	0.144
	1/3	0.943	0.937	0.929	0.920	0.991	0.902	0.904	0.144
	1/5	0.943	0.936	0.928	0.920	0.988	0.902	0.904	0.144
	1/7	0.943	0.936	0.928	0.919	0.987	0.902	0.904	0.144
	1/9	0.943	0.935	0.928	0.919	0.985	0.902	0.904	0.144
SC		0.899	0.863	0.948	0.947	0.980	0.876	0.899	0.210

Table 6.8: APFD values obtained from ICP_M with different $p_{[H][C]}$ values and ideal user model, ranging from 'extremely favours human expert's judgement (9)' to 'extremely favours coverage-based prioritisation ($\frac{1}{9}$)'.

tend to be less resistant to increasing error rate. On the other hand, test suites for which ICP is capable of making improvement over statement coverage-based prioritisation tend to produce more robust APFD values as the error rate increases. The test suite for `gzip` is capable of producing higher APFD values than statement coverage with error rates up to 0.45. Surprisingly, test suites for `schedule`, `sed`, `vim`, and two test suites for `space` are capable of producing higher APFD values than statement coverage with error rates up to 1.0, i.e. under the situation when the human expert always makes incorrect comparisons.

Figure 6.6 provides an explanation to this seemingly counter-intuitive phenomenon. Figure 6.6 contains following three boxplots. In each subplot, *Random* shows APFD of random ordering of test cases with no clustering; $RCRP$ (Random Clustering Random Prioritisation) shows APFD of randomised ICP with random clustering with $k = 14$ and $HCRP$ (Hierarchical Clustering Random Prioritisation) shows APFD of randomised ICP with hierarchical clustering. The solid horizontal line represents APFD value of the optimal ordering; the dotted horizontal line represents APFD value of the statement coverage-based prioritisation. There exists a common trend between all programs for which ICP produced successful improvement. For these programs, either the mean of $HCRP$ or the upper quartile observation of the box plot is higher than the APFD of statement coverage-based prioritisation. It can be concluded that the clustering with $k = 14$ works for the prioritisation of these programs and their test suites. Note that all the prioritisation is performed randomly for $HCRP$. Our conjecture is that, for these programs, any prioritisation technique that performs better than a purely random approach will eventually make an improvement over statement coverage-based prioritisation.

Table 6.9 confirms this conjecture, and provides an answer for **RQ4**. It compares the result of the suitability test for faults in AR and TBR sets of each program with the optimal ordering and the statement coverage-based prioritisation. Hierarchical Clustering/Statement Prioritisation (HCSP) represents the ICP with statement coverage prioritisation both for intra- and inter-cluster stage, combined with hierarchical agglom-

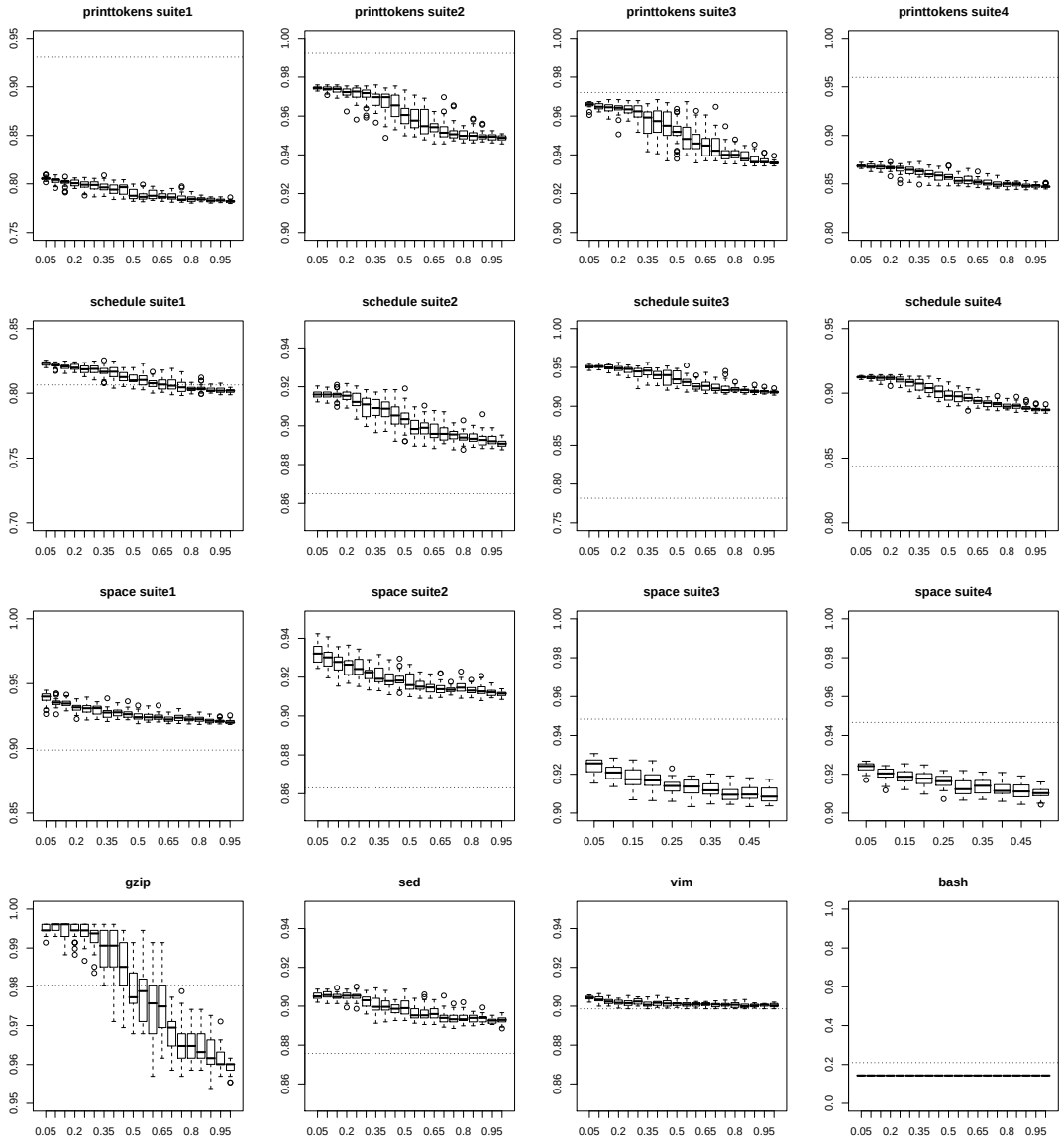

Figure 6.4: Boxplots of APFD values of ICP_S configuration with error rate ranging from 0.05 to 1.0 in steps of 0.05 on the x-axis. The y-axis shows the observed APFD metric values. For each error rate value, experiments are repeated 30 times to cater for the randomness in the error model. The horizontal dotted line shows the APFD value of statement coverage-based prioritisation. Surprisingly, for the test suites for which ICP_S showed an improvement in Table 6.7, the mean APFD values tend to stay above this dotted line, even when the error rate is above 0.5. In fact, with the exception of `schedule1` and `gzip`, even the error rate of 1.0 produces successful results for the test suites for which ICP_S showed an improvement in Table 6.7.

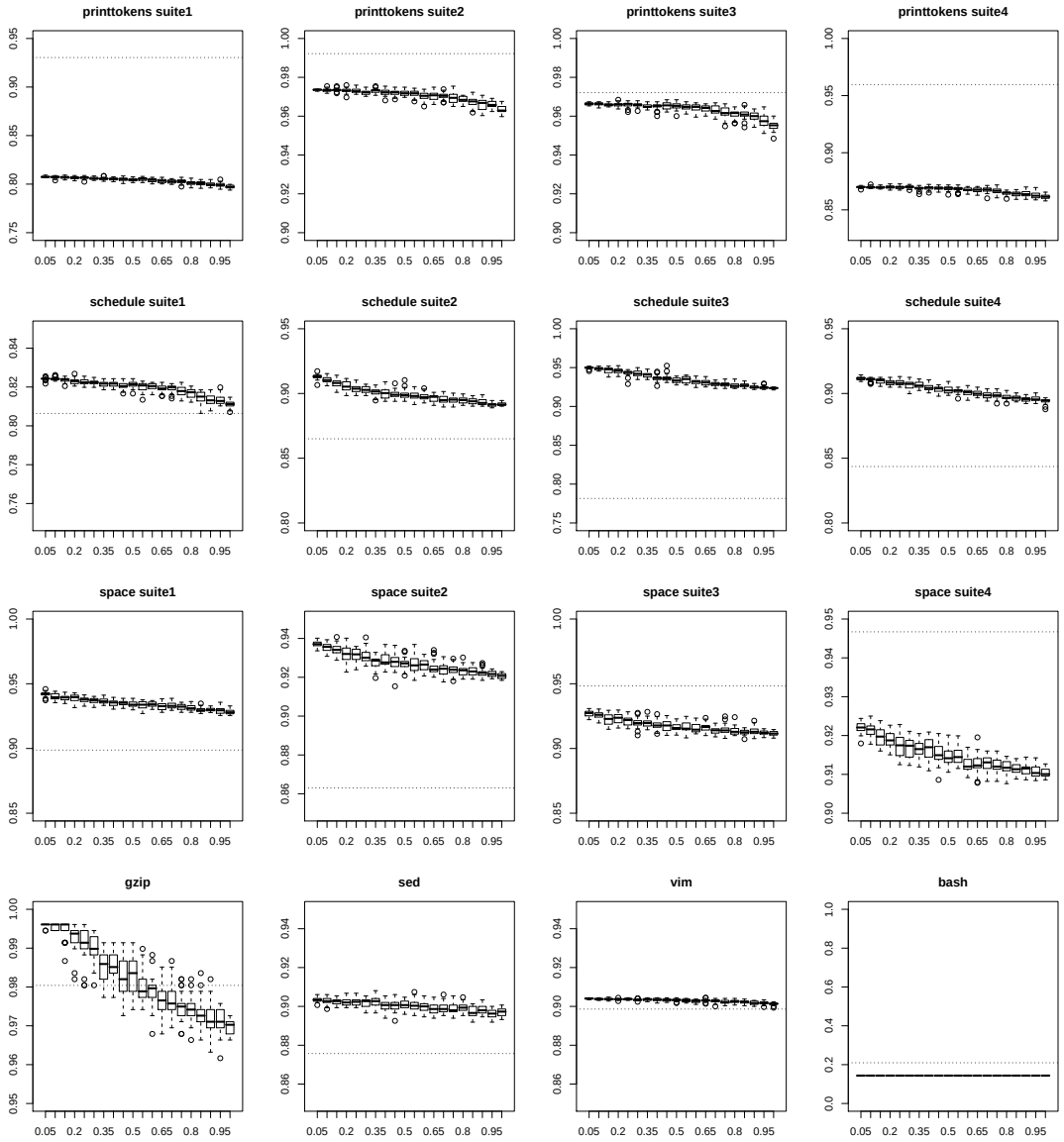

Figure 6.5: Boxplots of APFD values of ICP_M configuration with error rate ranging from 0.05 to 1.0 in steps of 0.05 on the x-axis. The y-axis shows the observed APFD metric values. The trend observed in Figure 6.4 continues. However, it can also be observed that the secondary prioritisation criteria (statement coverage) compliments human input. APFD values show smaller variances compared to Figure 6.4 and, in some cases, more tolerance in the presence of human error, for example, the case with test suite 1 of `schedule`.

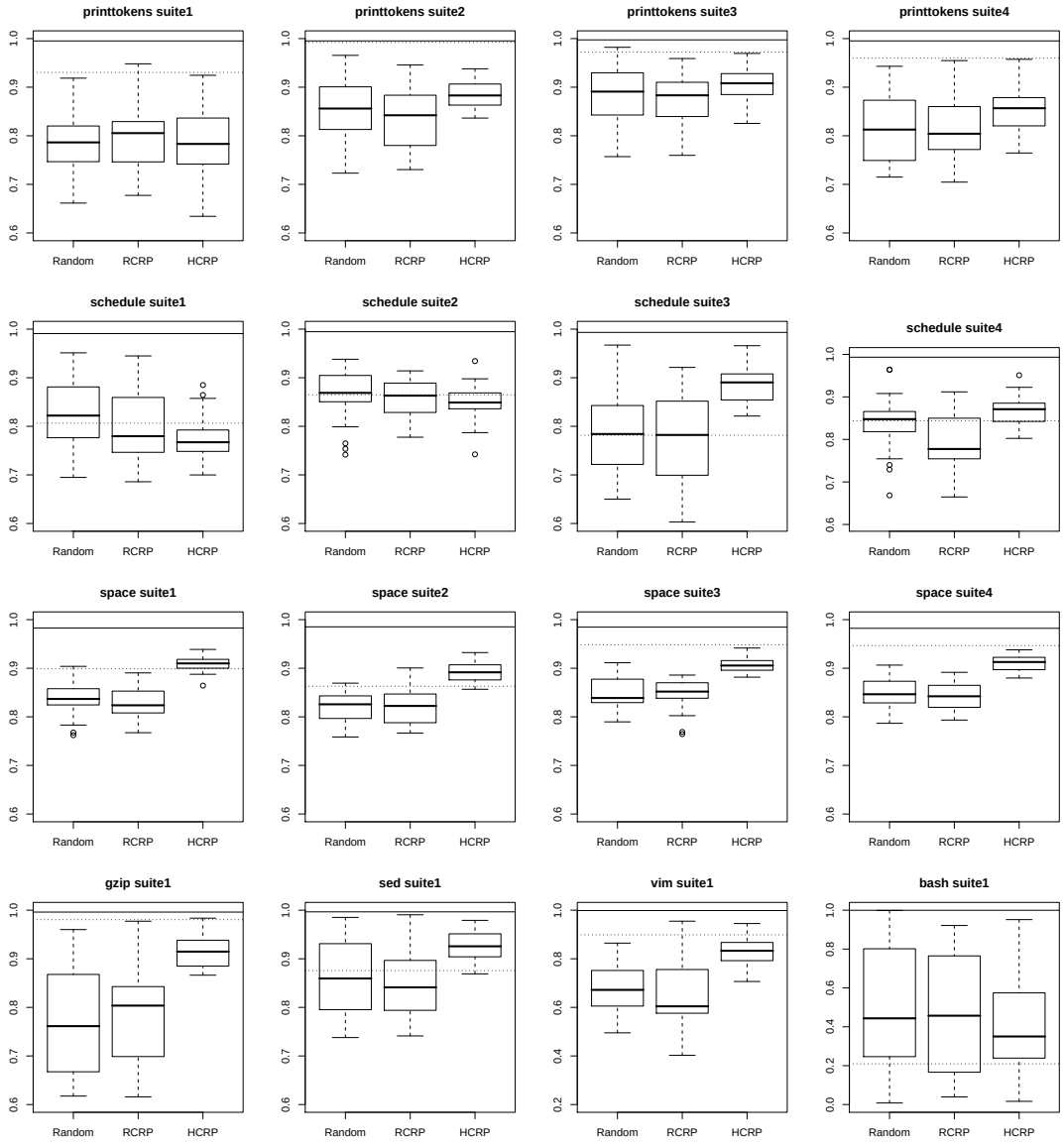

Figure 6.6: Boxplots of random prioritisation results. On the x-axis, 'Random' represents random prioritisation with no clustering; 'RCRP' represents random clustering and random ICP; 'HCRP' represents hierarchical clustering and random ICP. The y-axis shows the observed APFD metric values. For programs for which ICP performed well in Section 6.5.1, HCRP partially outperforms statement coverage-based prioritisation (represented by the dotted lines) even with random prioritisation.

erative clustering of $k = 14$. No Clustering/Statement Prioritisation (NCSP) represents traditional statement coverage-based prioritisation. Note that both configurations are deterministic and can be automated. For the faults in the AR set, both NCSP and HCSP are executed. If the result of HCSP is equal to or higher than that of NCSP, the specific pair of SUT and test suite is said to *pass* the suitability test. A *pass* means that the hierarchical clustering has a positive impact on statement coverage-based prioritisation. Therefore, we expect that replacing statement coverage-based prioritisation with techniques such as AHP will only improve the prioritisation of any SUT and test suite that passed suitability test. This expectation is then checked using the TBR set of faults. If ICP_M produces higher APFD than NCSP for a pair of (SUT, test suite) that passed suitability test, it can be said that the test produced a correct prediction. Since the aim of the suitability test is to avoid wasting human effort, false positive tests presents higher risk than false negative tests.

The results in Table 6.9 indicate the passed tests with grey background. For all 8 tests that passed, the subsequent experiments with faults in the TBR set confirm the result of suitability test. That is, ICP_M produces higher APFD than statement coverage-based prioritisation. There are two false negative results, marked with (*). Suite 2 of `schedule` and `vim` do not pass the suitability test, but ICP does produce higher APFD than NCSP. There is no false positive test result. The remaining 6 pairs of program/test suite do not pass the suitability test, and the subsequent experiments with faults in the TBR set confirm the correctness of the suitability test. In summary, **RQ4** is answered positively with 14 correct predictions (8 pass, 6 fail) out of 16 cases. The remaining two cases are comparatively harmless false negatives.

6.5.3 Limitations & Threats to Validity

Threats to internal validity concern the factors that might have affected the performance the proposed technique. The accuracy of execution trace data and fault detection data

Subject	printtokens				schedule			
Test Suite	1	2	3	4	1	2	3	4
OP	0.995	0.995	0.998	0.995	0.991	0.995	0.993	0.993
NCSP AR	0.936	0.997	0.998	0.965	0.899	0.974	0.922	0.949
HCSP AR	0.736	0.976	0.998	0.896	0.984	0.970*	0.972	0.986
NCSP TBR	0.915	0.993	0.953	0.949	0.831	0.880	0.854	0.883
ICP_M TBR	0.755	0.954	0.978	0.875	0.994	0.992	0.992	0.992
Subject	space				gzip	sed	vim	bash
Test Suite	1	2	3	4				
OP	0.983	0.985	0.985	0.982	0.996	0.997	0.998	0.999
NCSP AR	0.958	0.957	0.963	0.974	0.817	0.958	0.946	0.804
HCSP AR	0.964	0.977	0.941	0.939	0.831	0.959	0.890*	0.746
NCSP TBR	0.918	0.914	0.962	0.973	0.980	0.876	0.899	0.210
ICP_M TBR	0.966	0.982	0.956	0.944	0.996	0.905	0.905	0.144

Table 6.9: Results of the suitability test. NCSP is the traditional statement coverage-based prioritisation. HCSP is ICP with statement coverage prioritisation for both intra- and inter-cluster prioritisation. If HCSP performs no worse than NCSP, the test passes, i.e. ICP is expected to outperform NCSP with the faults in the TBR set. Cells with grey background show passed tests with correct prediction; cells with white background denote failed tests with correct prediction. The second test suite of `schedule` and the test suite of `vim` produce false negative results (denoted by *).

can have a significant impact on the performance. To address this, the execution traces were obtained using a widely used and well known compile/profiling tool, `gcc` and `gcov`. Fault detection data were obtained from SIR [41].

Threats to external validity concern the conditions that limit generalisation of the results. The novelty of the proposed technique lies in the combination of clustering and pair-wise comparisons. This study uses the agglomerative hierarchical clustering for the former and AHP for the latter, but other combinations of techniques may produce different results. Different combinations of techniques should be studied in order to address this concern. The representativeness of the test suites and subject programs is another primary concern. However, the study studies programs with sizes ranging from 412 LoC to over 100KLoc. When multiple test suites are available, four different test suites were randomly chosen to avoid any bias based on the choice of test suites.

6.6 Related Work

AHP has been widely adopted in various software engineering fields where the only meaningful prioritisation criteria is often the human preference. Karlsson et al. applied AHP to requirement prioritisation and empirically compared AHP with several different techniques [101]. Finnie et al. applied AHP to project management and prioritised productivity factors in software development [58]. Douligeris et al. applied AHP to evaluate the Quality-of-Service (QoS) in telecommunication networks [48].

Previous work in the test case management field studied how human involvement can improve the quality of test case prioritisation. Tonella et al. have successfully applied the Case-Base Ranking (CBR) machine learning technique to test case prioritisation [202]. CBR tries to learn the ordering between the test cases from the additional information it is given such as structural coverage. During the process, CBR presents the human tester with a pair of test cases and asks which one is more important. It combines the human

input with the information originally available, and produces an ordering of test cases. The empirical results show that prioritisation using CBR can outperform the existing coverage-based prioritisation techniques.

While this study starts from the same assumption (that human involvement can improve test case prioritisation), there are several significant differences from this previous work. First, this study introduces clustering to deal with the high cost of AHP. Without the reduction in effort, the cost of AHP has been considered inhibitive. Second, this study introduces the error rate in the user model, and thereby provides more realistic observation of the performance of the technique. Finally, the study introduces an automated assessment technique that determines whether human effort will be justified by the expected results.

6.7 Conclusions

This chapter introduced the use of clustering for human interactive test case prioritisation. Human interaction is achieved by using AHP, which is a widely used decision making tool that has been previously adopted by the Requirements Engineering community. Clustering is applied to reduce the number of pair-wise comparisons required by AHP, making it scalable to regression testing problems. The chapter introduced a hybrid Interleaved Clusters Prioritisation technique to combine these two techniques. The empirical studies show that the hybrid ICP algorithm can outperform the traditional coverage-based prioritisation for some programs, even when the human input is erroneous. The chapter also presented an automated suitability test that can accurately predict whether the hybrid ICP will make a positive improvement to the prioritisation of a specific pair of SUT and test suite, ensuring that no human effort is wasted on regression testing scenarios that are inappropriate for the technique.

Chapter 7

Conclusions

7.1 Summary of Achievements

The original aim of this thesis was to reformulate the existing regression testing techniques with respect to the issues of multi-objectiveness, test suite redundancy and the use of expert knowledge. Details of each objective were as follows:

1. To extend test case management problems and techniques by formulating multi-objective versions of both so that multi-constraint based real world complexity can be met; and

2. To introduce and evaluate a measurement of a form of "desirable" test suite redundancy, called test suite latency, to allow a systematic view on the issue of redundancy, and also to introduce an automated approach to latency enhancement, called test data augmentation, so that testers can enjoy a rich pool of novel test data at a low cost; and

3. To demonstrate how expert domain knowledge can be incorporated into existing test case management techniques so that testers can guide the automated regression testing process with human input without losing scalability.

7.1.1 Multi-objective Test Case Management

Most of the existing test case management techniques are single-objective; that is, the aim of the techniques is to maximise (or minimise) a singular property of the test suite. However, regression testing is a complex process that often involves multiple criteria and constraints. Existing techniques have tried to deal with the multiplicity of objectives using classical optimisation approach such as sequential optimisation of objectives, weighted sum of objectives, and combining two objectives in the form of ratio. While these approaches produce a solution for the problems, they do not provide any insights into the trade-offs between the multiple objectives, which are often more informative to the tester than a single solution.

The concept of Pareto-optimality has originally been introduced in economics [67], but now is extensively used in any domain that requires optimisation of multiple objectives. It allows us to observe the trade-offs between objectives in the form of a set of non-dominating solutions. This thesis instantiates test suite minimisation as a multi-objective optimisation problem using multiple objectives such as structural coverage, cost of execution and fault history. Theoretical analysis of existing techniques shows that multi-objective optimisation can provide more options for minimising test suites to testers so that they can perform more efficient regression testing under certain constraints. The empirical study confirms the theoretical observation with real world examples and presents visualisation of the multi-objective solution space.

7.1.2 Test Suite Latency & On-demand Regression Testing

One of the important issues in test case management is the question of the necessity of redundancy in test suites. In general, every novel test case brings more information about the correctness of SUT and, therefore, the redundancy is desirable. In reality, the tester has to balance the benefits of having a large and redundant test suite with the cost of managing such a test suite. One of the things that have been missing from

the discussion on test suite redundancy is how to measure and express the redundancy. Existing work in the literature often just identified the *minimised* subset of the test suites, and labelled the rest as redundant. However, this does not provide a clear view of the real redundancy as the quality of the *rest* of test suites can vary greatly.

This thesis presents a concept called *test suite latency*: a systematic measurement of redundancy in test suites with respect to specific set of testing goals. This allows the tester to observe the full, latent potential of the parts of test suites that are deemed to be redundant. This observation can lead the tester to form an informed decision about how much redundancy is cost-effective. The thesis also considers an automated strategy of enhancing test suite latency suing the existing test data as a starting point.

The concept of *on-demand* regression testing takes the automated latency enhancement strategy one step further. It is an approach to regression testing that is free from limitations of having too large test suites. In this approach, test suites only serve as a template from which additional test data can generated on-demand at a very low cost. This ensures that the tester enjoys a set of novel test data each time the technique is applied for regression testing, eliminating the risk of over-fitting the testing process to a small set of test data. It may also be used in parallel to existing test case management technique. Empirical study of real world examples shows that the proposed technique can be significantly cheaper than existing test data generation technique, while maintaining the fault detection capability.

7.1.3 Use of Expert Knowledge in Test Case Management

While automation of test case management is necessary due to the scalability issues, it is unrealistic to expect any human tester to accept the result of an automated technique without questions or objections. In fact, if the tester in question is very well experienced in the testing of a given SUT, it would certainly be beneficial to utilise the expert knowledge. However, the human tester is likely to wish to influence the decisions in test

case management for many other reasons, even if the outcome becomes sub-optimal in reality. Therefore, any test case management technique will benefit from allowing human testers to have their own input to the process, either for the effectiveness and efficiency of the technique or for the acceptance of the technique.

The main obstacle to the use of human knowledge in test case management is that human input is not only very costly but its expression also tends to be highly inconsistent. The key to the use of expert knowledge is to balance the benefits of human knowledge with its cost. This thesis presents an instantiation of test case prioritisation that uses human expert knowledge. The cost of human input is maintained to be manageable by using a clustering technique to reduce the problem size. For the test case prioritisation, the human tester only faces pair-wise comparison problems of a fixed and feasible size. The results of the prioritisation of the clustered test suites are then extrapolated to the original test suites using a novel prioritisation algorithm called Interleaved Clusters Prioritisation (ICP).

The proposed prioritisation approach is empirically evaluated not only with respect to ideal user model (a perfect tester who always knows which test case is more effective than the others) but also more realistic user models with varying error rates (a tester who will provide incorrect answers according to some given probability). The result of this comparative analysis reveals, somewhat surprisingly, that particular pairs of SUT and test suite are very resistant to human errors; in other words, clustering alone has a positive impact on the effectiveness of the ICP. This observation forms the basis of the automated test that determines whether expert knowledge based prioritisation will be more effective than existing techniques or not.

7.2 Summary of Future Work

7.2.1 Orchestrating Test Case Management with Test Data Generation

Automatic generation of test data has witnessed a burgeoning interest from the research community; a wide range of techniques such as meta-heuristic search techniques and symbolic execution has been successfully applied to the automatic test data generation problem. However, the study of connecting test data generation with test case management has been significantly overlooked so far. Since one technique aims to generate more test data and the other aims to reduce the effort required for executing all the test cases, there ought to exist a synergistic combination that can be obtained from orchestrating the two techniques; one can provide valuable feedback to the other and *vice versa*.

The latency enhancement strategy and test data augmentation technique introduced in this thesis can be viewed as one possible way of taking advantage of this potential synergy between two domains; the information obtained from one domain is used to reduce the cost of the other. However, it is possible to imagine a more sophisticated and richer collaboration between two domains. For example, test data generation can inform test case management techniques about the value of each test case estimated from the way it was generated. Similarly, test case management can guide future test data generation by providing the information about which parts of the program require more test data, or which test cases have been successful at detecting faults.

7.2.2 Non-functional Testing and Test Case Management

A majority of existing test case management techniques rely upon structural information about the SUT such as data flow analysis, CFG analysis, program slices and structural coverage. The impact that non-functional property testing will have on test case management techniques has not been fully studied. Existing techniques were able to map the problems in the test case management domain to well-formed abstract problems thanks

to the properties of structural information. For example, test suite minimisation could be mapped to the minimal hitting set problem or the set coverage problem precisely because the techniques were based on the concept of 'coverage'. Similarly, graph-walking approaches to test case selection were made possible because the changes between different versions were defined by structural difference in CFGs.

Suppose the goals of test case management techniques were focused on non-functional properties. What would be the minimised test suite that can test the power consumption of an embedded system? How would test cases be prioritised to achieve an efficient and effective stress testing of a web application? These questions remain largely unanswered and may require approaches that are significantly different from existing paradigms.

7.2.3 Industrial Scale Adaptation & Tool Support

While various test case management techniques have reached a level of maturity in the academic environment, the adoption by industry has been slow. This is evidenced by the fact that a majority of the literature on the subject depends on a small set of programs made available through Software Infrastructure Repository (SIR); so much so, in fact, that there may be a risk of over-fitting the entire subject to these programs, their test suites and fault information. This is partly due to the fact that any information related to faults in a software system is extremely sensitive to corporations. Nonetheless, empirical evaluation of test case management techniques in industrial scale is something that would carry immense value to academic researchers.

Closely related to this is the issue of tool support. Without readily available tools that implement test case management techniques, practical adoption will remain limited. One potential difficulty of providing tool support is the fact that, unlike unit testing for which there exists a series of frameworks based on the xUnit architecture, there is not a common framework for the regression testing process in general. The closest to a common ground for regression testing would be an Integrated Development Environment (IDE),

such as Eclipse, with which the xUnit architecture is already integrated successfully. A good starting point for test case management techniques may be the management framework of unit test cases, built upon xUnit architecture and IDEs.

Bibliography

[1] Cantata++ `http://www.ipl.com/products/tools/pt400.uk.php`.

[2] *IEEE Standard Glossary of Software Engineering Terminology.* IEEE Press, 10 Dec 1990.

[3] *The Oxford English Dictionary.* Oxford University Press, 2nd edition, April 2000.

[4] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London. Incremental regression testing. In *Proceedings of the International Conference on Software Maintenance (ICSM 1993)*, pages 348–357. IEEE Computer Society, September 1993.

[5] 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 2005)*, pages 402–411. ACM Press, May 2005.

[6] Ricardo Anido, Ana R. Cavalli, Luiz Paula Lima Jr, and Nina Yevtushenko. Test suite minimization for testing in context. *Software Testing, Verification and Reliability*, 13(3):141–155, 2003.

[7] Paolo Avesani, Cinzia Bazzanella, Anna Perini, and Angelo Susi. Facing scalability issues in requirements prioritization with machine learning techniques. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE 2005)*, pages 297–306. IEEE Computer Society, 2005.

[8] Thomas Ball. On the limit of control flow analysis for regression test selection. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 1998)*, pages 134–142. ACM Press, March 1998.

[9] André Baresel, David Wendell Binkley, Mark Harman, and Bogdan Korel. Evolutionary testing in the presence of loop–assigned flags: A testability transformation approach. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 108–118, July 2004.

[10] Samual Bates and Susan Horwitz. Incremental program testing using program dependence graphs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 384–396. ACM Press, January 1993.

[11] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. Automatic test case optimization: A bacteriologic algorithm. *IEEE Software*, 22(2):76–82, 2005.

[12] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. From genetic to bacteriological algorithms for mutation-based testing. *Software Testing, Verification, and Reliability*, 15(2):73–96, 2005.

[13] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving test suites for efficient fault localization. In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*, pages 82–91. ACM Press, 2006.

[14] Boris Beizer. *Software Testing Techniques.* International Thomson Computer Press, 2nd edition, 1990.

[15] P. Benedusi, A. Cmitile, and U. De Carlini. Post-maintenance testing based on path change analysis. In *Proceedings of the International Conference on Software*

Maintenance (ICSM 1988), pages 352–361. IEEE Computer Society Press, October 1988.

[16] S. Beydeda and V. Gruhn. Integrating white- and black-box techniques for class-level regression testing. In *Proceedings of the 25th IEEE International Computer Software and Applications Conference (COMPSAC 2001)*, pages 357–362, 2001.

[17] D. Binkley. Reducing the cost of regression testing by semantics guided test case selection. In *Proceedings of the International Conference on Software Maintenance (ICSM 1995)*, pages 251–260. IEEE Computer Society, 1995.

[18] J. Black, E. Melachrinoudis, and D. Kaeli. Bi-criteria models for all-uses test suite reduction. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 106–115, May 2004.

[19] Morris I. Bolsky and David G. Korn. *The New KornShell Command and Programming Language*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1995.

[20] Leonardo Bottaci. Instrumenting programs with flag variables for test data search by genetic algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 1337–1342. Morgan Kaufmann Publishers, July 2002.

[21] L. C. Briand, Y. Labiche, K. Buist, and G. Soccar. Automating impact analysis and regression test selection based on UML designs. In *Proceedings of the International Conference on Software Maintenance (ICSM 2002)*, pages 252–261. IEEE Computer Society, October 2002.

[22] L. C. Briand, Y. Labiche, and S. He. Automating regression test selection based on UML designs. *Journal of Information and Software Technology*, 51(1):16–30, 2009.

[23] Renee C. Bryce and Charles J. Colbourn. Test prioritization for pairwise interaction coverage. In *Proceedings of the ACM workshop on Advances in Model-Based Testing (A-MOST 2005)*, pages 1–7. ACM Press, 2005.

[24] Renée C. Bryce and Charles J. Colbourn. Prioritized interaction testing for pairwise coverage with seeding and constraints. *Journal of Information and Software Technology*, 48(10):960–970, 2006.

[25] Renée C. Bryce, Charles J. Colbourn, and Myra B. Cohen. A framework of greedy methods for constructing interaction test suites. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 146–155. ACM Press, May 2005.

[26] Renée C. Bryce and Atif M. Memon. Test suite prioritization by interaction coverage. In *Proceedings of the Workshop on Domain Specific Approaches to Software Test Automation (DOSTA 2007)*, pages 1–7. ACM, September 2007.

[27] Timothy Alan Budd. *Mutation analysis of program test data.* PhD thesis, Yale University, New Haven, CT, USA, 1980.

[28] Tsong Yueh Chen and Man Fai Lau. Dividing strategies for the optimization of a test suite. *Information Processing Letters*, 60(3):135–141, 1996.

[29] Yanping Chen, Robert L. Probert, and D. Paul Sims. Specification-based regression test selection with risk analysis. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research (CASCON 2002)*, page 1. IBM Press, October 2002.

[30] Yanping Chen, Robert L. Probert, and Hasan Ural. Regression test suite reduction using extended dependence analysis. In *Proceedings of the 4th International Workshop on Software Quality Assurance (SOQUA 2007)*, pages 62–69. ACM Press, September 2007.

[31] Yih Farn Chen, D. Rosenblum, and Kiem-Phong Vo. Testtube: A system for selective regression testing. In *Proveedings of the 16th International Conference on Software Engineering (ICSE 1994)*, pages 211–220. ACM PressM, May 1994.

[32] J.J. Chilenski and S.P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, Sep 1994.

[33] Carlos A. Coello Coello, David A. Van Veldhuizen, and Gary B. Lamont. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Kluwer Academic Publishers, New York, May 2002.

[34] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Exploiting constraint solving history to construct interaction test suites. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pages 121–132. IEEE Computer Society, 2007.

[35] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering*, 34(5):633–650, 2008.

[36] Y. Collette and P. Siarry. *Multiobjective Optimization: Principles and Case Studies*. Springer, Oxford, UK, August 2004.

[37] Kalyanmoy Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. Wiley, Chichester, UK, June 2001.

[38] Kalyanmoy Deb, Samir Agrawal, Amrit Pratab, and T. Meyarivan. A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II. In *Proceedings of the Parallel Problem Solving from Nature VI Conference*, pages 849–858, Paris, France, 2000. Springer. Lecture Notes in Computer Science No. 1917.

[39] D. Deng, P.C.-Y. Sheu, and T. Wang. Model-based testing and maintenance. In *Proceedings of the 6th IEEE International Symposium on Multimedia Software Engineering (MMSE 2004)*, pages 278–285, December 2004.

[40] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a junit testing environment. In *Proceedings of 15th International Symposium on Software Reliability Engineering (ISSRE 2004)*, pages 113–124. IEEE Computer Society Press, Nov 2004.

[41] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[42] Hyunsook Do, Siavash Mirarab Mirarab, Ladan Tahvildari, and Gregg Rothermel. An empirical study of the effect of time constraints on the cost-benefits of regression testing. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 71–82. ACM Press, November 2008.

[43] Hyunsook Do and Gregg Rothermel. A controlled experiment assessing test case prioritization techniques via mutation faults. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 411–420, 2005.

[44] Hyunsook Do and Gregg Rothermel. An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 141–151. ACM Press, November 2006.

[45] Hyunsook Do and Gregg Rothermel. On the use of mutation faults in empirical

assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9):733–752, 2006.

[46] Hyunsook Do and Gregg Rothermel. Using sensitivity analysis to create simplified economic models for regression testing. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 51–61. ACM Press, July 2008.

[47] Hyunsook Do, Gregg Rothermel, and Alex Kinneer. Prioritizing junit test cases: An empirical assessment and cost-benefits analysis. *Empirical Software Engineering*, 11(1):33–70, 2006.

[48] C. Douligeris and I.J. Pereira. A telecommunications quality study using the analytic hierarchy process. *IEEE Journal on Selected Areas in Communications*, 12(2):241–250, Feb 1994.

[49] S. Elbaum, D. Gable, and G. Rothermel. Understanding and measuring the sources of variation in the prioritization of regression test suites. In *Proceedings of the Seventh International Software Metrics Symposium (METRICS 2001)*, pages 169–179. IEEE Computer Society Press, April 2001.

[50] S. Elbaum, P. Kallakuri, A. Malishevsky, G. Rothermel, and S. Kanduri. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Software Testing, Verification, and Reliability*, 12(2), 2003.

[51] S. Elbaum, A.G. Malishevsky, and G. Rothermel. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, Feb 2002.

[52] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Control*, 12(3):185–210, 2004.

[53] Sebastian G. Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2000)*, pages 102–112. ACM Press, August 2000.

[54] Sebastian G. Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the International Conference on Software Engineering (ICSE 2001)*, pages 329–338. ACM Press, May 2001.

[55] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[56] Qurat-ul-ann Farooq, Muhammad Zohaib Z. Iqbal, Zafar I Malik, and Aamer Nadeem. An approach for selective state machine based regression testing. In *Proceedings of the 3rd International Workshop on Advances in Model-based Testing (A-MOST 2007)*, pages 44–52, New York, NY, USA, 2007. ACM.

[57] Robert Feldt, Richard Torkar, Tony Gorschek, and Wasif Afzal. Searching for cognitively diverse tests: Towards universal test diversity metrics. In *Proceedings of the 1st Workshop on Search-Based Software Testing (SBST 2008)*, pages 178 – 186. IEEE Computer Society Press, April 2008.

[58] Gavin R. Finnie, Gerhard Wittig, and Doncho I. Petkov. Prioritizing software development productivity factors using the analytic hierarchy process. *The Journal of Systems and Software*, 22(2):129–139, August 1993.

[59] K.F. Fischer. A test case selection method for the validation of software maintenance modifications. In *Proceedings of International Computer Software and*

Applications Conference, pages 421–426. IEEE Computer Society Press, November 1977.

[60] K.F. Fischer, F. Raji, and A. Chruscicki. A methodology for retesting modified software. In *Proceedings of the National Telecommunications Conference*, pages 1–6, November 1981.

[61] Marc Fisher II, Dalai Jin, Gregg Rothermel, and Margaret Burnett. Test reuse in the spreadsheet paradigm. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE 2002)*, pages 257–268, November, 2002. IEEE Computer Society.

[62] Phyllis G. Frankl, Stewart N. Weiss, and Cang Hu. All-uses vs mutation testing: An experimental comparison of effectiveness. *Journal of Systems Software*, 38:235–253, 1997.

[63] Gordon Fraser and Franz Wotawa. Property relevant software testing with model-checkers. *SIGSOFT Software Engineering Notes*, 31(6):1–10, 2006.

[64] Gordon Fraser and Franz Wotawa. Test-case prioritization with model-checkers. In *SE'07: Proceedings of the 25th conference on IASTED International Multi-Conference*, pages 267–272, Anaheim, CA, USA, 2007. ACTA Press.

[65] Y. Freund and R. Schapire. A short introduction to boosting. *Journal of Japanese Society for Artificial Intelligence*, 14(5):771–780, 1999.

[66] Yoav Freund, Raj Iyer, Robert E. Schapire, and Yoram Singer. An efficient boosting algorithm for combining preferences. In Jude W. Shavlik, editor, *Proceedings of the 15th International Conference on Machine Learning (ICML 1998)*, pages 170–178. Morgan Kaufmann Publishers, July 1998.

[67] D. Fudenberg and J. Tirole. *Game Theory*, chapter 1. MIT Press, 1983.

[68] Matthew J. Gallagher and V.Lakshmi Narasimhan. Adtest: A test data generation suite for ada software systems. *IEEE Transactions on Software Engineering*, 23(8):473–484, 1997.

[69] M. R. Garey and D. S. Johnson. *Computers and Intractability: A guide to the theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[70] Todd Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. In *Proceedings of the 20th International Conference on Software Engineering (ICSE 1998)*, pages 188–197. IEEE Computer Society Press, April 1998.

[71] Rajiv Gupta, Mary Jean Harrold, and Mary Lou Soffa. An approach to regression testing using slicing. In *Proceedings of the International Conference on Software Maintenance (ICSM 1992)*, pages 299–308. IEEE Computer Society Press, November 1992.

[72] Michael Harder, Jeff Mellen, and Michael D. Ernst. Improving test suites via operational abstraction. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 60–71. IEEE Computer Society, 2003.

[73] M. Harman and P. McMinn. A theoretical and empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2007)*, pages pp. 73–83. ACM Press, July 2007.

[74] Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, and Joachim Wegener. The impact of input domain reduction on search-based test data generation. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE 2007)*, pages 155–164. ACM, September 2007.

[75] Mark Harman and Joachim Wegener. Evolutionary testing. In *Genetic and Evolutionary Computation (GECCO)*, New York, July 2002.

[76] M. J. Harrold and M. L. Soffa. Interprocedual data flow testing. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 158–167. ACM Press, December 1989.

[77] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, 1993.

[78] Mary Jean Harrold. Testing evolving software. *The Journal of Systems and Software*, 47(2–3):173–181, 1999.

[79] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, and Steven Spoon. Regression test selection for Java software. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001)*, pages 312–326. ACM Press, October 2001.

[80] Mary Jean Harrold, David S. Rosenblum, Gregg Rothermel, and Elaine J. Weyuker. Empirical studies of a prediction model for regression test selection. *IEEE Transactions on Software Engineering*, 27(3):248–263, 2001.

[81] Mary Jean Harrold and Mary Lou Soffa. An incremental approach to unit testing during maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM 1998)*, pages 362–367. IEEE Computer Society Press, October 1988.

[82] Mary Jean Harrold and Mary Lou Soffa. An incremental data flow testing tool. In *Proceedings of the 6th International Conference on Testing Computer Software (ICTCS 1989)*, May 1989.

[83] J. Hartmann and D. J. Robson. Revalidation during the software maintenance phase. In *Proceedings of the International Conference on Software Maintenance (ICSM 1989)*, pages 70–80. IEEE Computer Society Press, October 1989.

[84] J. Hartmann and D. J. Robson. Retest-development of a selective revalidation prototype environment for use in software maintenance. In *Proceedings of the International Conference on System Sciences*, volume 2, pages 92–101. IEEE Computer Society Press, January 1990.

[85] Jean Hartmann and David J. Robson. Techniques for selective revalidation. *IEEE Software*, 7(1):31–36, 1990.

[86] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*, chapter 14.3.12 Hierarchical clustering, pages 272–280. Springer, 2001.

[87] Robert Hierons, Mark Harman, and Chris Fox. Branch-coverage testability transformation for unstructured programs. *The Computer Journal*, 48(4):421–436, 2005.

[88] J. R. Horgan and S. London. Data flow coverage and the C language. In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*, pages 87–97. ACM Press, October 1991.

[89] J.R. Horgan and S.A London. ATAC: A data flow coverage testing tool for c. In *Proceedings of the Symposium on Assessment of Quality Software Development Tools*, pages 2–10. IEEE Computer Society Press, June 1992.

[90] Shan-Shan Hou, Lu Zhang, Tao Xie, Hong Mei, and Jia-Su Sun. Applying interface-contract mutation in regression testing of component-based software. In *Proc. 23rd IEEE International Conference on Software Maintenance (ICSM 2007)*, pages 174–183, October 2007.

[91] Shan-Shan Hou, Lu Zhang, Tao Xie, and Jia-Su Sun. Quota-constrained test-case prioritization for regression testing of service-centric systems. In *Proc. IEEE International Conference on Software Maintenance (ICSM 2008)*, October 2008.

[92] Hwa-You Hsu and Alessandro Orso. MINTS: A general framework and tool for supporting test-suite minimization. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 419–429. IEEE Computer Society, May 2009.

[93] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering (ICSE 1994)*, pages 191–200. IEEE Computer Society Press, May 1994.

[94] D. Jeffrey and N. Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on Software Engineering*, 33(2):108–123, 2007.

[95] D. Jeffrey and Neelam Gupta. Test suite reduction with selective redundancy. In *Proceedings of the 21st IEEE International Conference on Software Maintenance 2005 (ICSM'05)*, pages 549–558, September 2005.

[96] Dennis Jeffrey and Neelam Gupta. Test case prioritization using relevant slices. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC 2006)*, pages 411–420, Washington, DC, USA, September 2006. IEEE Computer Society.

[97] David S. Johnson. Approximation algorithms for combinatorial problems. In *Proceedings of the 5th annual ACM Symposium on Theory of Computing (STOC 1973)*, pages 38–49. ACM Press, May 1973.

[98] B.F. Jones, H.-H. Sthamer, and D.E. Eyres. Automatic structural testing using genetic algorithms. *The Software Engineering Journal*, 11(5):299–306, 1996.

[99] James A. Jones and Mary Jean Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. In *Proceedings of International Conference on Software Maintenance (ICSM 2001)*, pages 92–101. IEEE Computer Society Press, November 2001.

[100] Garrett Kent Kaminski and Paul Ammann. Using logic criterion feasibility to reduce test set size while guaranteeing fault detection. In *Proceedings of International Conference on Software Testing, Verification, and Validation 2009 (ICST 2009)*, pages 356–365. IEEE Computer Society, 2009.

[101] Joachim Karlsson, Claes Wohlin, and Björn Regnell. An evaluation of methods for prioritizing software requirements. *Information & Software Technology*, 39(14-15):939–947, 1998.

[102] Jung-Min Kim and Adam Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, pages 119–129. ACM Press, May 2002.

[103] Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test application frequency. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 126–135. ACM Press, June 2000.

[104] Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test application frequency. *Software Testing, Verification, and Reliability*, 15(4):257–279, 2005.

[105] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[106] B. Korel, G. Koutsogiannakis, and L.H. Tahat. Application of system models in regression test suite prioritization. In *Proceedings of IEEE International Conference on Software Maintenance 2008 (ICSM 2008)*, pages 247–256, October 2008.

[107] B. Korel, L. Tahat, and B. Vaysburg. Model based regression test reduction using dependence analysis. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 214–225. IEEE Computer Society, October 2002.

[108] B. Korel, L.H. Tahat, and M. Harman. Test prioritization using system models. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 559–568, September 2005.

[109] Bogdan Korel, Mark Harman, S. Chung, P. Apirukvorapinit, and R. Gupta. Data dependence based testability transformation in automated test generation. In *Proceedings of the 16th International Symposium on Software Reliability Engineering (ISSRE 05)*, pages 245–254. IEEE Computer Society Press, November 2005.

[110] Bogdan Korel, George Koutsogiannakis, and Luay H. Tahat. Model-based test prioritization heuristic methods and their evaluation. In *Proceedings of the 3rd international workshop on Advances in Model-based Testing (A-MOST 2007)*, pages 34–43. ACM Press, July 2007.

[111] David Korn and Kiem phong Vo. SFIO: Safe/Fast String/File IO. In *Proceedings of the Summer Usenix Conference 1991*, pages 235–256, 1991.

[112] R. Krishnamoorthi and S. A. Sahaaya Arul Mary. Factor oriented requirement coverage based system test case prioritization of new and regression test cases. *Information and Software Technology*, 51(4):799–808, 2009.

[113] David Chenho Kung, Jerry Gao, Pei Hsia, Jeremy Lin, and Yasufumi Toyoshima. Class firewall, test order, and regression testing of object-oriented programs. *Journal of Object–Oriented Programming*, 8(2):51–65, May 1995.

[114] Kiran Lakhotia, Mark Harman, and Phil McMinn. A multi-objective approach to search-based test data generation. In *Proceedings of the 9th Annual Conference on Genetic and evolutionary Computation (GECCO 2007)*, pages 1098–1105. ACM Press, July 2007.

[115] J. Laski and W. Szermer. Identification of program modifications and its applications in software maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM 1992)*, pages 282–290. IEEE Computer Society Press, November 1992.

[116] Man F. Lau and Yuen T. Yu. An extended fault class hierarchy for specification-based testing. *ACM Transactions on Software Engineering Methodology*, 14(3):247–276, 2005.

[117] Y. Le Traon, T. Jeron, J.-M. Jezequel, and P. Morel. Efficient object-oriented integration and regression testing. *IEEE Transactions on Reliability*, 49(1):12–25, March 2000.

[118] J. A. N. Lee and Xudong He. A methodology for test selection. *Journal of Systems and Software*, 13(3):177–185, 1990.

[119] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. Efficient unit test case minimization. In *Proceedings of the 22nd IEEE/ACM international conference on Automated Software Engineering (ASE 2007)*, pages 417–420. ACM Press, November 2007.

[120] David Leon and Andy Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *Proceedings*

of the IEEE International Symposium on Software Reliability Engineering (ISSRE 2003), pages pp. 442–456. IEEE Computer Society Press, November 2003.

[121] H. K. N. Leung and L. White. Insight into regression testing. In *Proceedings of Interntional Conference on Software Maintenance (ICSM 1989)*, pages 60–69. IEEE Computer Society Press, October 1989.

[122] H. K. N. Leung and L. White. Insights into testing and regression testing global variables. *Journal of Software Maintenance*, 2(4):209–222, 1990.

[123] H. K. N. Leung and L. White. A study of integration testing and software regression at the integration level. In *Proceedings of the International Conference on Software Maintenance (ICSM 1990)*, pages 290–301. IEEE Computer Society Press, November 1990.

[124] H. K. N. Leung and L. White. A cost model to compare regression test strategies. In *Proceedings of the International Conference on Software Maintenance (ICSM 1991)*, pages 201–208. IEEE Computer Society Press, October 1991.

[125] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[126] Zheng Li, Mark Harman, and Robert M. Hierons. Search Algorithms for Regression Test Case Prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, 2007.

[127] Feng Lin, Michael Ruth, and Shengru Tu. Applying safe regression test selection techniques to java web services. In *Proceedings of the International Conference on Next Generation Web Services Practices (NWESP 2006)*, pages 133–142. IEEE Computer Society, September 2006.

[128] Yu Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava: an automated class mutation system. *Software Testing, Verification, and Reliability*, 15(2):97–133, 2005.

[129] A. Malishevsky, G. Rothermel, and S. Elbaum. Modeling the cost-benefits tradeoffs for regression testing techniques. In *Proceedings of the International Conference on Software Maintenance (ICSM 2002)*, pages 230–240. IEEE Computer Society Press, 2002.

[130] Alexey G. Malishevsky, Joseph R. Ruthruff, Gregg Rothermel, and Sebastian Elbaum. Cost-cognizant test case prioritization. Technical Report TR-UNL-CSE-2006-0004, Department of Computer Science and Engineering, University of Nebraska-Lincoln, March 2006.

[131] M. Marré and A. Bertolino. Using spanning sets for coverage testing. *IEEE Transactions on Software Engineering*, 29(11):974–984, November 2003.

[132] Eliane Martins and Vanessa Gindri Vieira. Regression test selection for testable classes. *Lecture Notes in Computer Science : Dependable Computing - EDCC 2005*, 3463/2005:453–470, 2005.

[133] Scott McMaster and Atif Memon. Call-stack coverage for gui test suite reduction. *IEEE Transactions on Software Engineering*, 34(1):99–115, 2008.

[134] Scott McMaster and Atif M. Memon. Call stack coverage for test suite reduction. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 539–548, Washington, DC, USA, 2005. IEEE Computer Society.

[135] Scott McMaster and Atif M. Memon. Fault detection probability analysis for coverage-based test suite reduction. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'07)*. IEEE Computer Society, October 2007.

[136] Phil McMinn. IGUANA: Input generation using automated novel algorithms. A plug and play research tool. Technical Report CS-07-14, Department of Computer Science, University of Sheffield, 2007.

[137] Phil McMinn, David Binkley, and Mark Harman. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Transactions on Software Engineering Methodology*, 18(3), 2009. To appear July 2009.

[138] Phil McMinn, Mark Harman, David Binkley, and Paolo Tonella. The species per path approach to search-based test data generation. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 06)*, pages 13–24. ACM Press, July 2006.

[139] Philip McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

[140] C.C. Michael, G. McGraw, and M.A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, December 2001.

[141] W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223–226, 1976.

[142] Siavash Mirarab and Ladan Tahvildari. A prioritization approach for software test cases based on bayesian networks. In *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering*, pages 276–290, March/April 2007.

[143] Siavash Mirarab and Ladan Tahvildari. An empirical study on bayesian network-based approach for test case prioritization. In *Proceedings of International Conference on Software Testing, Verification and Validation*, pages 278–287. IEEE Computer Society, April 2008.

[144] Henry Muccini, Marcio Dias, and Debra J. Richardson. Reasoning about software architecture-based regression testing through a case study. In *Proceedings of the 29th International Computer Software and Applications Conference (COMPSAC 2005)*, volume 2, pages 189–195. IEEE Computer Society, 2005.

[145] Henry Muccini, Marcio Dias, and Debra J. Richardson. Software-architecture based regression testing. *Journal of Systems and Software*, 79(10):1379–1396, October 2006.

[146] Heinz Mühlenbein and Gerhard Paaß. From recombination of genes to the estimation of distributions I. Binary Parameters. In *Proceedings of the 4th International Conference on Parallel Problem Solving from Nature (PPSN IV)*, pages 178–187. Springer-Verlag, September 1996.

[147] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 89–100. ACM Press, June 2007.

[148] A. Jefferson Offutt, Z. Jin, and Jie Pan. The dynamic domain reduction approach to test data generation. *Software Practice and Experience*, 29(2):167–193, January 1999.

[149] A. Jefferson Offutt, Jie Pan, Kanupriya Tewary, and Tong Zhang. An experimental evaluation of data flow and mutation testing. *Software Practice and Experience*, 26(2):165–176, February 1996.

[150] J. Offutt, J. Pan, and J. Voas. Procedures for reducing the size of coverage-based test sets. In *Proceedings of the 12th International Conference on Testing Computer Software*, pages 111–123, June 1995.

[151] A. Orso, M. J. Harrold, D. S. Rosenblum, G. Rothermel, M. L. Soffa, and H. Do. Using component metadata to support the regression testing of component-based

software. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2001)*, November 2001.

[152] Alessandro Orso, Hyunsook Do, Gregg Rothermel, Mary Jean Harrold, and David S. Rosenblum. Using component metadata to regression test component-based software: Research articles. *Software Testing, Verification, and Reliability*, 17(2):61–94, 2007.

[153] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2004)*, pages 241–251. ACM Press, October/November 2004.

[154] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization*. Courier Dover Publications, 1998.

[155] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9(4):263–282, December 1999.

[156] O. Pilskalns, G. Uyan, and A. Andrews. Regression testing uml designs. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM 2006)*, pages 254–264, September 2006.

[157] Hartmut Pohlheim and Joachim Wegener. Testing the temporal behavior of real-time software modules using extended evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999)*, pages 1795–1802. Morgan Kaufmann, July 1999.

[158] Xiao Qu, Myra B. Cohen, and Gregg Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *Proceedings of the*

ACM International Symposium on Software Testing and Analysis (ISSTA 2008), pages 75–86. ACM Press, July 2008.

[159] Xiao Qu, Myra B. Cohen, and Katherine M. Woolf. Combinatorial interaction regression testing: A study of test case generation and prioritization. In *Proceedings of IEEE International Conference on Software Maintenance (ICSM 2007)*, pages 255–264. IEEE Computer Society Press, October 2007.

[160] C. Ramey and B. Fox. *Bash Reference Manual*. O'Reilly and Associates, Sebastopol, CA, 2.2 edition, 1998.

[161] John Rice. *Mathematical Statistics and Data Analysis*. Duxbury Press, 2nd Ed. 1995.

[162] D.S. Rosenblum and E.J. Weyuker. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Transactions on Software Engineering*, 23(3):Page(s):146 – 156, March 1997.

[163] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia. The impact of test suite granularity on the cost-effectiveness of regression testing. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, pages 130–140. ACM Press, May 2002.

[164] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and X. Qiu. On test suite composition and cost-effective regression testing. *ACM Transactions on Software Engineering and Methodology*, 13(3):277–331, July 2004.

[165] G. Rothermel, M. Harrold, J. Ronne, and C. Hong. Empirical studies of test suite reduction. *Software Testing, Verification, and Reliability*, 4(2):219–249, December 2002.

[166] G. Rothermel, M. Harrold, J. von Ronne, C. Hong, and J. Ostrin. Experiments to assess the costbenefits of test-suite reduction. Technical Report GIT-99-29, College of Computing, Georgia Institute of Technology, 1999.

[167] G. Rothermel and M. J. Harrold. A safe, efficient algorithm for regression test selection. In *Proceedings of International Conference on Software Maintenance (ICSM 2003)*, pages 358–367. IEEE Computer Society Press, September 1993.

[168] Gregg Rothermel. *Efficient, Effective Regression Testing Using Safe Test Selection Techniques*. PhD thesis, University of Clemson, May 1996.

[169] Gregg Rothermel and Mary Jean Harrold. A framework for evaluating regression test selection techniques. In *Proceedings of the 16th International Conference on Software Engineering (ICSE 1994)*, pages 201–210. IEEE Computer Society Press, 1994.

[170] Gregg Rothermel and Mary Jean Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 1994)*, pages 169–184. ACM Press, August 1994.

[171] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, August 1996.

[172] Gregg Rothermel and Mary Jean Harrold. Experience with regression test selection. *Empirical Software Engineering: An International Journal*, 2(2):178–188, 1997.

[173] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[174] Gregg Rothermel and Mary Jean Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24(6):401–419, 1998.

[175] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. Regression test selection for C++ software. *Software Testing, Verification and Reliability*, 10(2):77–109, June 2000.

[176] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings of International Conference on Software Maintenance (ICSM 1998)*, pages 34–43. IEEE Computer Society Press, November 1998.

[177] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *Proceedings of International Conference on Software Maintenance (ICSM 1999)*, pages 179–188. IEEE Computer Society Press, August 1999.

[178] Gregg Rothermel, Roland J. Untch, and Chengyun Chu. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.

[179] Matthew Rummel, Gregory M. Kapfhammer, and Andrew Thall. Towards the prioritization of regression test suites with data flow information. In *Proceedings of the 20th Symposium on Applied Computing (SAC 2005)*. ACM Press, March 2005.

[180] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[181] M. Ruth, Sehun Oh, A. Loup, B. Horton, O. Gallet, M. Mata, and Shengru Tu. Towards automatic regression test selection for web services. In *Proceedings of the*

31st International Computer Software and Applications Conference (COMPSAC 2007), pages 729–736. IEEE Computer Society Press, July 2007.

[182] M. Ruth and Shengru Tu. Concurrency issues in automating rts for web services. In *Proceedings of the IEEE International Conference on Web Services (ICWS 2007)*, pages 1142–1143. IEEE Computer Society Press, July 2007.

[183] Michael Ruth and Shengru Tu. A safe regression test selection technique for web services. In *Proceedings of the 2nd International Conference on Internet and Web Applications and Services (ICIW 2007)*, pages 47–47. IEEE Computer Society Press, May 2007.

[184] T.L. Saaty. *The Analytic Hierarchy Process, Planning, Piority Setting, Resource Allocation*. McGraw-Hill, New York, NY, USA, 1980.

[185] R. Sagarna and J. A. Lozano. Scatter search in software testing, comparison and collaboration with estimation of distribution algorithms. *European Journal of Operational Research*, 169(2):392–412, March 2006.

[186] Ramón Sagarna, Andrea Arcuri, and Xin Yao. Estimation of distribution algorithms for testing object oriented software. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2007)*, pages 438–444. IEEE Computer Society Press, September 2007.

[187] Sreedevi Sampath, Renee C. Bryce, Gokulanand Viswanath, Vani Kandimalla, and A. Gunes Koru. Prioritizing user-session-based test cases for web applications testing. In *Proceedings of the 1st International Conference on Software Testing Verification and Validation (ICST 2008)*, pages 141–150. IEEE Computer Society, April 2008.

[188] Patrick J. Schroeder and Bogdan Korel. Black-box test reduction using input-output analysis. *SIGSOFT Software Engineering Notes*, 25(5):173–177, 2000.

[189] Mark Sherriff, Mike Lake, and Laurie Williams. Prioritization of regression tests using singular value decomposition with empirical change records. In *Proceedings of the The 18th IEEE International Symposium on Software Reliability (ISSRE 2007)*, pages 81–90, Washington, DC, USA, 2007. IEEE Computer Society.

[190] M. Skoglund and P. Runeson. A case study of the class firewall regression test selection technique on a large scale distributed software system. In *Proceedings of International Symposium on Empirical Software Engineering (ISESE 2005)*, pages 74–83, November 2005.

[191] Adam Smith, Joshua Geiger, Gregory M. Kapfhammer, and Mary Lou Soffa. Test suite reduction and prioritization with call trees. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. ACM Press, November 2007.

[192] Adam M. Smith and Gregory M. Kapfhammer. An empirical study of incorporating cost into test suite reduction and prioritization. In *Proceedings of the 24th Symposium on Applied Computing (SAC 2009)*. ACM Press, March 2009.

[193] H. Srikanth, L. Williams, and J. Osborne. System test case prioritization of new and regression test cases. In *Proceedings of International Symposium on Empirical Software Engineering*, pages 64–73, November 2005.

[194] Amitabh Srivastava and Jay Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 97–106. ACM Press, July 2002.

[195] Harmen Sthamer. *The Automatic Generation of Software Test Data Using Genetic Algorithms*. PhD thesis, University of Glamorgan, 1996.

[196] F. Szidarovsky, M. E. Gershon, and L. Dukstein. *Techniques for multiobjective decision making in systems management*. Elsevier, New York, 1986.

[197] A. B. Taha, S. M. Thebaut, and S. S. Liu. An approach to software fault localization and revalidation based on incremental data flow analysis. In *Proceedings of the International Computer Software and Applications Conference (COMPSAC 1989)*, pages 527–534. IEEE Computer Society Press, September 1989.

[198] Sriraman Tallam and Neelam Gupta. A concept analysis inspired greedy algorithm for test suite minimization. *SIGSOFT Software Engineering Notes*, 31(1):35–42, 2006.

[199] A. Tarhini, H. Fouchal, and N. Mansour. Regression testing web services-based applications. In *Proceedings of ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 2006)*, pages 163–170. IEEE Computer Society Press, August 2006.

[200] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, May 2002.

[201] Marouane Tlili, Stefan Wappler, and Harmen Sthamer. Improving evolutionary real-time testing. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (GECCO 2006)*, pages 1917–1924. ACM Press, July 2006.

[202] Paolo Tonella, Paolo Avesani, and Angelo Susi. Using the case-based ranking methodology for test case prioritization. In *Proceedings of the 22nd International Conference on Software Maintenance (ICSM 2006)*, pages 123–133. IEEE Computer Society, July 2006.

[203] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *Proceedints of International Symposium on Software Testing and Analysis (ISSTA 1998)*, pages 73–81. ACM Press, March 1998.

[204] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *Proceedings of the 13th IEEE International Conference on Automated Software Engineering (ASE 1998)*, pages 285–288. IEEE Computer Society Press, October 1998.

[205] Nigel Tracey, John Clark, and Keith Mander. The way forward for unifying dynamic test-case generation: The optimisation-based approach. In *International Workshop on Dependable Computing and Its Applications (DCIA 1998)*, pages 169–180. IFIP, January 1998.

[206] Nigel Tracey, John Clark, John McDermid, and Keith Mander. *A search-based automated test-data generation framework for safety-critical systems*, pages 174–213. Springer-Verlag, New York, NY, USA, 2002.

[207] Boris Vaysburg, Luay H. Tahat, and Bogdan Korel. Dependence analysis in reduction of requirement based test suites. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 107–111. ACM Press, July 2002.

[208] F. Vokolos and P. Frankl. Pythia: A regression test selection tool based on text differencing. In *Proceedings of the International Conference on Reliability Quality and Safety of Software Intensive Systems*, May 1997.

[209] F.I. Vokolos and P.G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 1998)*, pages 44–53. IEEE Computer Society Press, November 1998.

[210] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. Time aware test suite prioritization. In *Proceedings of the International*

Symposium on Software Testing and Analysis (ISSTA 2006), pages 1–12. ACM Press, July 2006.

[211] Joachim Wegener, André Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, December 2001.

[212] Joachim Wegener and M. Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, 15(3):275 – 298, November 1998.

[213] Joachim Wegener, Harmen Sthamer, Bryan F. Jones, and David E. Eyres. Testing real-time systems using genetic algorithms. *Software Quality*, 6:127–135, 1997.

[214] L. White, H. Almezen, and S. Sastry. Firewall regression testing of gui sequences and their interactions. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2003)*, pages 398–409. IEEE Computer Society Press, September 2003.

[215] L. White and B. Robinson. Industrial real-time regression testing and analysis using firewalls. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, pages 18–27. IEEE Computer Society Press, September 2004.

[216] L. J. White and H. K. N. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *Proceedings of International Conference on Software Maintenance (ICSM 1992)*, pages 262–271. IEEE Computer Society Press, September 1992.

[217] L. J. White, V. Narayanswamy, T. Friedman, M. Kirschenbaum, P. Piwowarski, and M. Oha. Test manager: A regression testing tool. In *Proceedings of Interna-*

tional Conference on Software Maintenance (ICSM 1993), pages pages 338–347. IEEE Computer Society Press, September 1993.

[218] Lee White, Khaled Jaber, Brian Robinson, and Václav Rajlich. Extended firewall for regression testing: an experience report. *Journal of Software Maintenance and Evolution*, 20(6):419–433, 2008.

[219] W. Eric Wong, Joseph R. Horgan, Saul London, and Hira Agrawal Bellcore. A study of effective regression testing in practice. In *Proceedings of the 8th International Symposium on Software Reliability Engineering (ISSRE 1997)*, pages 264–275. IEEE Computer Society, 1997.

[220] W. Eric Wong, Joseph R. Horgan, Saul London, and Aditya P. Mathur. Effect of test set minimization on fault detection effectiveness. *Software Practice and Experience*, 28(4):347–369, April 1998.

[221] W. Eric Wong, Joseph R. Horgan, Aditya P. Mathur, and Alberto Pasquini. Test set size minimization and fault detection effectiveness: A case study in a space application. *The Journal of Systems and Software*, 48(2):79–89, October 1999.

[222] Ye Wu and J. Offutt. Maintaining evolving component-based software with UML. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR 2003)*, pages 133–142, March 2003.

[223] Guoqing Xu and Atanas Rountev. Regression test selection for AspectJ software. In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*, pages 65–74. IEEE Computer Society, May 2007.

[224] S. S. Yau and Z. Kishimoto. A method for revalidating modified programs in the maintenance phase. In *Proceedings of International Computer Software and Applications Conference (COMPSAC 1987)*, pages 272–277. IEEE Computer Society Press, October 1987.

[225] Shin Yoo and Mark Harman. Pareto efficient multi-objective test case selection. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2007)*, pages 140–150. ACM Press, July 2007.

[226] Shin Yoo, Mark Harman, Paolo Tonella, and Angelo Susi. Clustering test cases to achieve effective & scalable prioritisation incorporating expert knowledge. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2009)*, pages 201–211. ACM Press, July *to appear*.

[227] Shin Yoo, Mark Harman, and Shmuel Ur. Measuring and improving latency to avoid test suite wear out. In *Proceedings of the Interntional Conference on Software Testing, Verification and Validation Workshop (ICSTW 2009)*, pages 101–110. IEEE Computer Society Press, April 2009. Best paper award winner; To Appear.

[228] Yanbing Yu, James A. Jones, and Mary Jean Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *Proceedings of the International Conference on Software Engineering (ICSE 2008)*, pages 201–210. ACM Press, May 2008.

[229] Andreas Zeller. Yesterday, my program worked. today, it does not. why? *SIGSOFT Software Engineering Notes*, 24(6):253–267, 1999.

[230] Lu Zhang, Shan-Shan Hou, Chao Guo, Tao Xie, and Hong Mei. Time-aware test-case prioritization using Integer Linear Programming. In *Proceedings of the International Conference on Software Testing and Analysis (ISSTA 2009)*, pages 212–222. ACM Press, July 2009.

[231] Jianjun Zhao, Tao Xie, and Nan Li. Towards regression test selection for aspect-oriented programs. In *Proceedings of the 2nd Workshop on Testing Aspect-Oriented Programs (WTAOP 2006)*, pages 21–26. ACM Press, July 2006.

[232] Jiang Zheng, Brian Robinson, Laurie Williams, and Karen Smiley. Applying regression test selection for COTS-based applications. In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*, pages 512–522. ACM Press, 2006.

[233] Jiang Zheng, Brian Robinson, Laurie Williams, and Karen Smiley. A lightweight process for change identification and regression test selection in using cots components. In *Proceedings of the 5th International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS 2006)*, pages 137–146. IEEE Computer Society, 2006.

[234] Jiang Zheng, Laurie Williams, and Brian Robinson. Pallino: automation to support regression test selection for COTS-based applications. In *Proceedings of the 22nd IEEE/ACM international conference on Automated Software Engineering (ASE 2007)*, pages 224–233. ACM Press, November 2007.

[235] Jiang Zheng, Laurie Williams, Brian Robinson, and Karen Smiley. Regression test selection for black-box dynamic link library components. In *Proceedings of the 2nd International Workshop on Incorporating COTS Software into Software Systems: Tools and Techniques (IWICSS 2007)*, pages 9–14. IEEE Computer Society, January 2007.