Mutation-Optimised Subdomains for Test Data Generation and Program Analysis

Matthew Timothy Patrick

Doctor of Philosophy

University of York

Department of Computer Science

September 2013

For my devoted wife, Yalan.

Abstract

Software testing is an important part of the development process - it consumes a large proportion of the labour resources required to produce a working program. Yet it is not usually possible to show that a program is completely free from faults. Instead, techniques are applied to assess the effectiveness of software testing; they provide confidence in its adequacy and act as a benchmark for its improvement. One such technique (mutation analysis) uses small changes in the program code to simulate actual faults. Mutation analysis has been shown to be more stringent than other testing techniques and a good predictor of the real fault-finding capability of a test suite.

This thesis introduces new techniques for identifying, evolving and selecting input subdomains that can be sampled at random to produce efficient test suites which achieve a high level of mutation adequacy, and so are expected to be efficient at finding faults. Previous research into software testing has focussed on producing suites of individual test cases. This thesis represents the first attempt to optimise subdomains for each parameter to the program under test. The resulting subdomains can easily be comprehended by a human test engineer, so may be used to provide information about the software under test and design further highly efficient test suites.

Table of Contents

TABLE OF CONTENTS

List of Figures

LIST OF FIGURES

LIST OF FIGURES

Acknowledgements

I would like to thank my supervisor, John A. Clark for providing me the opportunity to undertake this PhD and for his support, encouragement and guidance throughout its completion. I would also like to thank my second supervisors, Manuel Oriol and Rob Alexander for their invaluable suggestions and feedback on my work.

I am grateful to David White, Kamran Ghani and Kiran Lakhotia for inspiring me at the beginning of my PhD and helping me to get started on the right track with my research. I would like to thank Matt Beech for his advice and assistance in preparing for my viva. I also appreciate the various fruitful discussions about mutation testing I have had with Yue Jia and Mike Papadakis. Yue Jia in particular has been a great source of motivation for me to finish this thesis.

The achievements of my PhD would not have been possible without the continued support of my parents who made it possible and encouraged me to pursue my passion in computer science. Finally, I would like to thank my wife Yalan for her endless patience and love both during the enjoyable and the difficult times of this project.

Declaration

I, Matthew Timothy Patrick, declare that all the work in this thesis is my own, except where attributed to another author. Some ideas and figures have appeared in the following publications:

Matthew Patrick, Manuel Oriol and John A. Clark. MESSI: Mutant Evaluation by Static Semantic Interpretation.

Proceedings: Mutation 2012

Matthew Patrick, Rob Alexander, Manuel Oriol and John A. Clark. Using Mutation Analysis to Evolve Subdomains for Random Testing.

Proceedings: Mutation 2013

Matthew Patrick, Rob Alexander, Manuel Oriol and John A. Clark. Efficient Subdomains for Random Testing.

Proceedings: SSBSE 2013

Matthew Patrick, Rob Alexander, Manuel Oriol and John A. Clark. Selecting Highly Efficient Sets of Subdomains for Mutation Adequacy.

Proceedings: APSEC 2013

Matthew Patrick, Rob Alexander, Manuel Oriol and John A. Clark. Probability-Based Semantic Interpretation of Mutants.

Proceedings: Mutation 2014

Matthew Patrick, Rob Alexander, Manuel Oriol and John A. Clark. Subdomain-Based Test Data Generation.

Journal of Systems and Software [under review]

Chapter 1

Introduction

Software testing is the process of exercising software so as to check whether it functions correctly [110]. Software testers look for errors by generating and executing test inputs according to the specification (black-box) or internal structure (white-box) of the software. Black-box and white-box techniques cannot typically be used to test software exhaustively [110]. Programs have too many paths and potential input values to test them all. As a result, black-box testing is often performed using randomly chosen inputs and white-box testing is considered adequate once a certain set of structural components are covered by the test suite.

This thesis introduces a new technique that combines elements from black-box and white-box testing. Regions of the input domain (known as subdomains) are optimised, such that random test cases sampled from them are more likely to detect faults efficiently. Test suites produced using this technique are typically more effective than in traditional random testing and the tester is able to choose values from within the subdomains that are representative of actual inputs.

Significant human effort is required to evaluate test cases [105]. Existing techniques generate input values that are scattered across the entire input domain. McMinn et al. [105] suggest that test inputs can be evaluated more quickly if their values are closer together. Subdomains make testing less expensive because they reduce the range of inputs testers must consider and decrease the number of test cases that they have to evaluate. Subdomains may be sampled randomly to produce efficient test suites, or the information gained through their optimisation can be used by testers to construct even more powerful test suites themselves.

1.1 Software Testing

Software testing is used to provide confidence in the correctness of software [110]. Inadequate testing may result in software products that are unsatisfactory or unsafe. For example, during the first Gulf war, a rounding error in a Patriot surface-to-air missile battery led to it failing to identify an incoming Iraqi Scud missile [17]. As a result, 28 American soldiers were killed and many more were injured. In 2012, an undisclosed fault in a high frequency trading program caused a financial services firm (Knight Capital Group) to lose $440 million in 30 minutes [71]. The firm lost 75% of the value of its stock in two days and was sold to another trading company four months later. Software failures can be very expensive to developers and/or users. Testing is a key part of the software development lifecycle because it helps to ensure that programs function as intended.

Developers make various kinds of mistakes, ranging from incorrectly interpreting the specification through to underestimating the usage requirements or just plain typographic mistakes [16]. Developer mistakes are known as faults. More broadly, a fault is defined as an incorrect step, process or data definition within a program [77]. Faults lead to errors in software behaviour. Software testing is the process of executing a program with the intent of finding as many of its errors as possible [110]. Software is executed with a set of inputs and conditions known as a test case. A collection of test cases is called as a test suite. Software testing therefore involves three stages: Designing test suites capable of finding faults, executing of test cases and determining whether the output is correct [16].

Software is particularly difficult to test because it is intangible, unique and highly specialised to a particular purpose [99]. It is estimated that between 30-90% of the labour resources required to produce a working program are spent on software testing [16]. For example, Microsoft employ approximate one test engineer for every developer [126]. Yet, despite this investment many faults are often missed. The Java Compatibility Kit [144] is an extensive test suite developed by Sun for the Java Development Kit (JDK), yet there are still thousands of additional JDK bug reports in Sun's bug database [124]. Most programs have too many paths to show that they are all correct [43]. The problem of testing is therefore two-fold: first it is very expensive; second it is often incomplete.

Automatic software test data generation tools have been developed in an attempt to save time and money, as well as improve the standard of testing [96]. Test outputs can be checked automatically for run-time errors, assertion violations and incorrect outputs. Any program, process or body of data that specifies the expected outcome of a test suite is known as an oracle [16]. The quality of testing depends upon the quality of the oracle. Automated testing is best suited to stable, well understood software whose specification is accurate and whose functionality does not change frequently [45]. Unfortunately software typically does not start in this state, even if it does get there eventually.

Considerable manual effort is required to construct effective test oracles and then update them as and when the software changes [45]. Many automated testing initiatives have failed due to poor planning and misunderstandings of the limitations involved [111]. It is also difficult to automate some oracles because they rely on expert insight. Weyuker [152] suggests as an example that a financial specialist may be able to distinguish whether a company's assets are correctly reported as $1,000,000 or $1,100,000, but it would be difficult to specify this formally because even the expert does not know what the exact result should be. Manual testing is often performed to a poor standard because it is repetitive and boring [110]. An important objective is to minimise the number of test cases that testers have to evaluate manually to ensure a high standard of testing.

It is dangerous to evaluate test quality purely in terms of the number of faults that are found because finding many faults may indicate good test data or poor software. Test data should be considered to provide sufficient confidence in the quality of software if it achieves some independent adequacy criterion. Zhu et al [160] describe three categories of adequacy criteria: Structural criteria emphasise the need to exercise particular components in the program code (statements, branches, paths etc.); Error-based criteria ensure the input domain is covered thoroughly (e.g. by partition testing); Fault-based criteria, the focus of this thesis, acknowledge testing as adequate if it detects a range of artificially introduced mistakes in the software. Fault-based criteria are considered superior because they measure the actual ability of a test suite to find faults (of course this depends on how representative the faults are). I use a fault-based criterion to help testers focus on regions of the input domain that are more likely to reveal faults.

1.2 Mutation Analysis

Mutation Analysis is a fault-based testing technique based around the idea that small syntactic changes can be used to simulate actual faults. The concept was first introduced in 1971 by Richard Lipton [94]. Since then, there have been over 400 research papers and at least 36 software tools have been developed [78][79]. Mutation analysis is supported by the *competent programmer hypothesis* (experienced programmers produce programs that are either correct or very close to being correct) [24] and the *coupling effect hypothesis* (test suites capable of detecting all the simple faults in a program can also detect most of the more complex ones) [41]. Developers may understand how the program should behave and have made a small mistake in its implementation, or have a slight misconception about the intended behaviour and carry it through to the implementation. In either case, small syntactic changes are sufficient to represent most faults [115].

A *mutant* is a copy of the original program that has had a small syntactic change (known as a *mutation*) made to its logical and arithmetic constructs. Mutations are typically applied one at a time. For example, in Figure 1.1, the greater-than inequality of line 1 has been replaced with a greater-than-or-equal-to inequality. A mutant is said to be *killed* by input values that cause it to output a different result to the original program. The mutant in Figure 1.1 is killed when the value of 'a' is equal to 10 and the value of 'b' is not equal to zero. Mutation analysis evaluates test suites according to the mutants they kill. A test suite is considered to be effective if it kills a large proportion of the mutants.

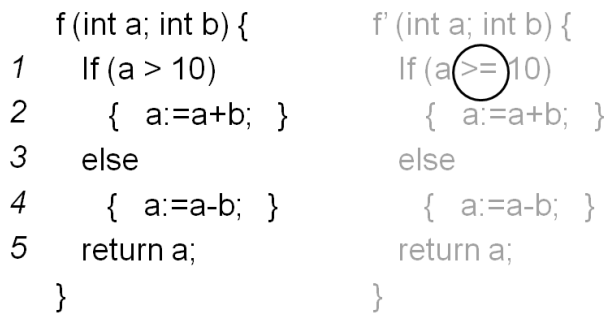


Figure 1.1: A Simple Syntactic Mutation

Mutation analysis can be applied iteratively to help improve the quality and efficiency of a test suite (see Figure 1.2). The proportion of mutants killed by the test suite is known as its mutation score (see Equation 1.1). This value may be used to indicate weaknesses, since if the test suite fails to kill some of the mutants, it is also likely to miss actual faults. Test cases are typically added or removed in an attempt to kill the remaining mutants as efficiently as possible. Some mutants cannot be killed, since they function equivalently to the original program for every possible input; they are typically removed from the calculation, so that mutation score is correctly scaled between 0 and 1. Mutation analysis is applied and the test suite is improved until it is able to kill all (or most of) the non-equivalent mutants. The intention is that by improving the test suite against a set of mutants, its ability to detect actual faults is also improved.

$$\text{Mutation score} = \frac{\text{number of mutants killed}}{\text{number of non-equivalent mutants}} \tag{1.1}$$

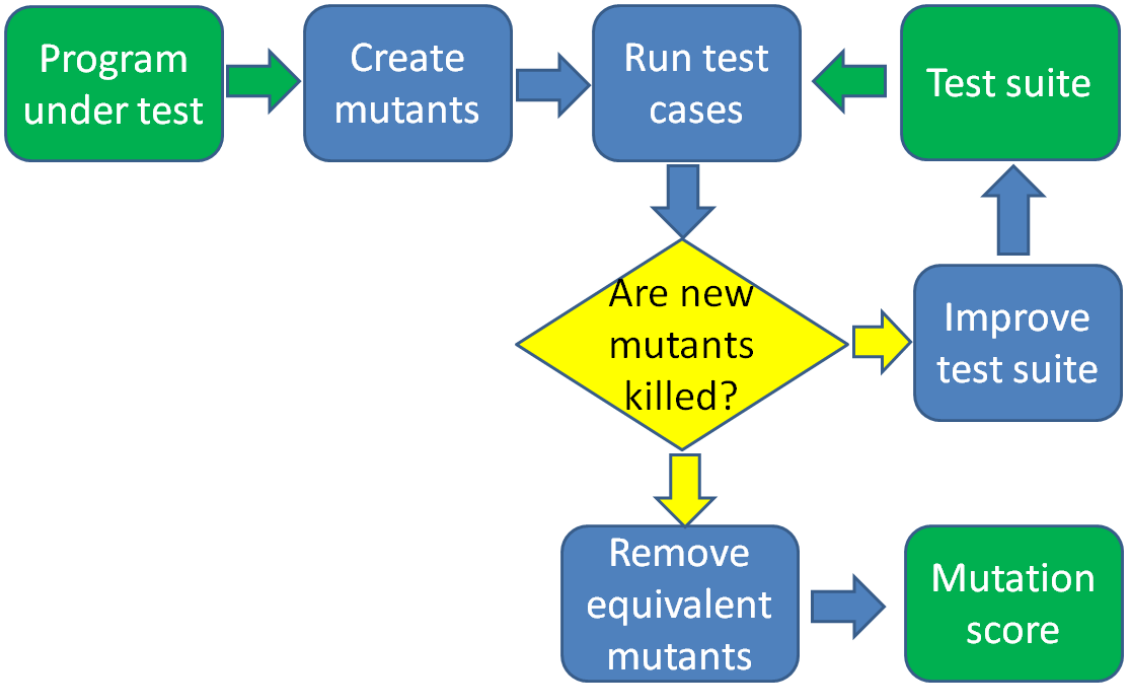


Figure 1.2: The Process of Mutation Analysis, adapted from [121]

1.3 The Problem Addressed by this Thesis

Test data generation techniques have proven to be effective at achieving a high mutation score [128][51][65]. Yet they provide little feedback about the relationship between input values and fault detection. For example, when a test case is generated to kill the mutant in Figure 1.1 using the value 10 for 'a' and 'b', it is unclear whether this value is significant, or if other values might also kill the same mutant. Test suites produced automatically contain test cases that are obscure and difficult to understand [105]. As a result, testers may become separated from the test generation process and this can lead to poor quality testing.

Subdomains offer a solution to this problem. To have a high probability of killing the mutant in Figure 1.1, the subdomain for 'a' must be small and include the value 10, whereas the subdomain for 'b' can be much larger. It is clear from optimising subdomains for this mutant that the value 10 is important for 'a', but not for 'b'. Looking at the code, the tester can see that this value occurs in the branch condition on line 1. When applied to programs, subdomains can highlight execution scenarios a tester may have missed. This helps to reduce the human effort involved in testing and increase the tester's ability to find faults.

1.4 Aims and Objectives

The main aim of this thesis is to make software testing more efficient by introducing new techniques for identifying and evaluating efficient sets of input subdomains for test data generation. I do this through a combination of random testing, mutation analysis, metaheuristic optimisation and static analysis.

The detailed objectives of this thesis are as follows:

1. Apply subdomain optimisation so as to reduce the number of test cases and input regions that must be evaluated by software testers.

2. Specialise subdomains at killing different groups of mutants, so that they can cover the input domain more effectively and efficiently.

3. Investigate the capability of static program analysis to reduce the number of subdomains that are required to test software effectively.

1.5 Thesis Structure

Chapter 2 surveys the literature on mutation analysis and test data generation. It begins by comparing the relative strengths and weaknesses of mutation analysis and structural testing, then discusses some ways to overcome the limitations of mutation analysis. It then describes methods for making random test data generation more efficient, before briefly discussing some techniques for static analysis.

Chapter 3 introduces a technique for optimising efficient subdomains of test input (one for each input parameter). I achieve this through the use of mutation analysis, random testing and metaheuristic optimisation. The resulting subdomains can be sampled at random to produce test suites that achieve a higher mutation score than unoptimised random testing.

Chapter 4 extends the previous technique to work with multiple sets of subdomains and then selects highly efficient sets from those that have been optimised. I achieve this through the use of a more advanced optimisation technique and automated program transformation. Test suites sampled from the resulting subdomains achieve a higher mutation score than single sets of subdomains.

Chapter 5 introduces two techniques for assessing mutant similarity. Their aim is to identify mutants that are likely to produce the most effective subdomains, thereby reducing the number of subdomains that are produced. These mutants reduce the computational expense involved with subdomain optimisation without significantly affecting the mutation score.

Chapter 6 concludes the thesis and provides directions for future research.

Chapter 2

Literature Review

2.1 Mutation Analysis vs. Structural Criteria

Mutation analysis and structural test criteria can both be used to measure the effectiveness of a test suite. Mutation analysis counts the mutants the test suite kills, whereas structural criteria count the structural components the test suite exercises. Structural criteria only require components to be reached by the program execution, whereas mutation analysis requires each mutation to affect the output [122]. There are various forms of weak/firm mutation that allow faults to be revealed at an earlier point in the execution [79]. Yet, this thesis focuses on classical (strong mutation) because test suites that kill all the mutants under strong mutation also kill all the mutants under weak/firm mutation.

One advantage of mutation analysis over structural criteria is that it can be tailored to use different mutation operators. Operators have been developed for C and Java through to SQL and AspectJ [79], addressing the unique features of each language. For example, the JavaScript keyword 'var' is added or removed to simulate mistakes in declaring variables locally or globally [108]. There are also operators that address common misunderstandings in C [34]. Depending on the program under test, different operators are likely to be more effective - data processing benefits more from operators that change values, whereas altering branch conditions is more useful for control programs. The author is not aware of any research that addresses this directly, but it is taken into account indirectly by differences in the program statements where each operator can be applied.

There are two main forms of structural criteria: control-flow and data flow. The rest of this section explores each of these in turn, in order to provide more details about the differences between mutation analysis and structural criteria:

2.1.1 Control-Flow Criteria

Statement coverage is the simplest control flow criterion, but it is not possible to exercise all the statements if some (dead) code cannot be reached [154]. Branch coverage ensures that every reachable statement is exercised [70]. Mutation analysis subsumes branch coverage, as long as each branch contains at least one statement that can be mutated [122]. On the other end of the scale, path coverage checks whether every possible sequence of decisions is covered, but it is infeasible for most programs (particularly those containing loops). Mutation analysis is a feasible (though sometimes expensive) technique and (in contrast to statement coverage) it also checks the coverage of branch conditions [122].

Techniques have been proposed to make path coverage more feasible, such as limiting the length of paths or the depth of loops that are explored [56] [127]. McCabe [102] suggested not investigating a path if it is made up of sub-paths that have already been explored. More typical however, is the use of branch condition criteria to make branch coverage stronger. Condition coverage checks whether each condition in a branch has evaluated as both True and False [110]. Modified Condition Decision Coverage (MC/DC) checks for condition and branch coverage, but also requires that each condition has an independent effect on which branch is taken [69]. Mutation analysis subsumes MC/DC, as long as it includes operators that modify every branch condition [122]. This is because, for a mutation of a branch condition to have an effect on the output, it must change the condition evaluation such that it results in a different branch being taken.

Mutation analysis has also been shown to be more stringent than control flow coverage criteria in practice. On Rolls Royce engine control modules that had already been tested to the military standards of statement and MC/DC coverage, mutation analysis revealed 50% and 32% new failures respectively [11]. Similarly, test data generated with mutation analysis found more faults than prime path coverage (path coverage ignoring loops) on 11 Java programs [93].

2.1.2 Data-Flow Criteria

Structural criteria can also take into account the flow of data through a program by considering the definitions and uses of its variables [50]. The *all-uses* strategy covers at least one path from the definition of every variable to each of its uses, whilst the *all-du-paths* strategy covers every such path. A distinction may also be made between the use of variables for comparison in a predicate and in computing new values. For example, the *all-p-uses/some-c-uses* strategy covers every predicate use, but if none exist it ensures that at least one computational use has been covered [136]. Mutation analysis was also shown to find more faults than all-uses on 11 Java programs [49][93], but there have been no other experimental comparisons with data-flow criteria. It is therefore still an open question whether any of the more advanced criteria are stricter or as strict as mutation analysis.

2.1.3 Summary

Mutation analysis and structural criteria provide measurements of test suite effectiveness. Improving the evaluation of any of these metrics is likely to help the tester find more faults. Significant computational effort can be wasted in structural analysis, because it is not possible to know whether the selected path will be feasible before trying to exercise it [76]. A similar problem occurs in mutation analysis, since some mutants are semantically equivalent to the original program and cannot be killed no matter how much computational effort is expended. Overall, however, improvements in mutation analysis has been shown to be more closely correlated to finding faults than structural criteria.

The main advantage of mutation analysis is that its operators are based on faults that programmers are actually likely to make. Experimentally and in practice, mutants have been shown to be correlated to actual faults. For example, selected mutation operators provided a better indication of fault detection ability than manually seeded faults with the Siemens suite [6]. In a civil nuclear program, 85% of the errors produced by mutants were also produced by real faults [39]. Research has therefore shown mutation analysis to be more stringent than other testing criteria and a good predictor of real fault finding capability.

2.2 Solutions for Overcoming the Limitations of Mutation Analysis

Mutation analysis is effective at evaluating the fault-finding capability of test suites, but there are some problems that must be overcome before mutation analysis can be used successfully to evaluate the effectiveness of test suites. The remainder of this section considers solutions to the following four problems:

1. **Equivalent mutants:** It is an undecidable problem whether mutants are semantically equivalent to the original program. Considerable human effort is often involved in identifying and removing equivalent mutants.

2. **Too many mutants:** Mutation operators produce a large number of mutants when they are applied at every possible location in the program code. It is computationally expensive to run the test suite on all the mutants.

3. **Unrealistic mutants:** Mutants are generated as small syntactic changes, but a small change in syntax can have a large effect on semantics. Some mutants do not represent faults that are likely to be found in practice.

4. **Difficult to kill mutants:** Mutation analysis can be used to improve test suites, but manually it is labour-intensive to create test cases and provide an oracle. In the worst case, a separate test case is needed for each mutant.

2.2.1 Equivalent Mutants

A mutated statement will have no effect on the program's output if it is performed on code that is never triggered or its effect is cancelled out at a later point [57]. Such mutants are described as being *equivalent* to the original program. Equivalent mutants cannot be distinguished by any test data, but still have an effect on the mutation score. It is difficult to predict how many mutants of a program will be equivalent, as experiments show this to vary from 5% up

to 54% [109][140]. Equivalent mutants may pose a significant threat to the validity of mutation analysis, so it is important to identify and remove them from consideration before using mutation in test data analysis [121].

2.2.1.1 Detecting Equivalent Mutants by Hand

People generally judge equivalent mutants correctly 80% of the time by hand, but this process is very time consuming [1]. Equivalence can be observed for some mutations by looking only at the statement that has changed (e.g. inserting a post-increment operator into a return statement). For the majority of programs [109], it is necessary to consider the program state immediately prior to the point of mutation (e.g. changing a plus to a minus will have no effect if the value to be added is zero). Some equivalent mutants, however, can only be identified by tracing their execution all the way to the output. It is worth considering any technique that can reduce the effort involved in detecting equivalent mutants.

One optimisation involves the use of program slices [72]. A program slice only contains the relevant components for a particular state and point in code [149]. This makes it easier to tell by hand whether a mutant is equivalent. Slices can be created statically from a control flow graph, or dynamically from an execution trace. Dynamic slices are typically smaller, but require carefully chosen input values if they are to include all the relevant components. Amorphous slices relax the syntax of the original program to reveal its semantics more clearly. This requires further processing, but as human effort is very expensive, anything that can be done computationally to reduce this may be worthwhile.

Figure 2.1 shows an example of a static (syntax-preserving) and an amorphous (not syntax-preserving) slice on a simple function, 'f'. The static slice removes line 5 from the original program because 'b' has no effect on the output. This makes the code easier to read, as there are now only two variables that must be considered. We also know any mutation that occurs on line 5 is equivalent. The amorphous slice also removes variable 'c' from the code by replacing it at the point where it is used by its value (20). If the division on line 6 is replaced with a multiplication, the value becomes 500, so this mutant is clearly not equivalent.

Original Program:

```
1 int f()
2 {
3   int a,b,c;
4   a = 100;
5   b = 50;
6   c = a/5;
7   a = a + c;
8   return a;
9 }
```

Static Slice:

```
1 int f()
2 {
3   int a,c;
4   a = 100;
5
6   c = a/5;
7   a = a + c;
8   return a;
9 }
```

Amorphous Slice:

```
1 int f()
2 {
3   int a;
4
5
6
7   a = 100+20;
8   return a;
9 }
```

Figure 2.1: Example of Static and Amorphous Program Slicing

2.2.1.2 Detecting Equivalent Mutants Automatically

Compiler optimisation can be used to identify equivalent mutants without human intervention. Mutants are equivalent if they are transformed into the same intermediate form as the original program [116]. Compiler optimisation techniques only identified 10% of the equivalent mutants in an experiment with 15 programs (see Figure 2.2). Yet, even though they are not as effective as manual analysis, compiler optimisation techniques are worthwhile because of their low cost.

Useful compiler optimisation techniques include *dead code detection*, *constant propagation* and *invariant propagation*. Mutations that occur within dead code cannot be reached, so must be equivalent. Mutations are also equivalent if they take the absolute value of a constant that is greater than zero or replace one variable with another of the same value. Of these techniques, invariant propagation was shown to be the most successful (see Figure 2.2).

Automated constraint satisfiers can be used to determine whether a mutation propagates its effect to the output [119]. If the set of constraints cannot be satisfied, the mutation is presumed to be equivalent. In an experiment involving 11 subject programs, this technique found 47.63% of the equivalent mutants (see Table 2.1). Although this is better than compiler optimisation, it is still worse than manual detection. Offutt [121] suggested that the undetected mutants can be safely ignored, but mutation analysis seems ineffective as measure of adequacy when it involves so much imprecision.

Table 2.1: Equivalent Mutants Detected by Constraint Solving [119]

Program	Lines of Code	Equivalent Mutants	Percentage Detected
Bsearch	20	27	70.37%
Bubble	11	35	68.57%
Cal	29	236	15.67%
Euclid	11	24	75.00%
Find	28	75	84.00%
Insert	14	46	69.57%
Mid	16	13	23.08%
Pat	17	61	47.54%
Quad	10	31	12.90%
Trityp	28	109	73.39%
Warshall	11	35	62.86%
		Mean Detected:	54.81%

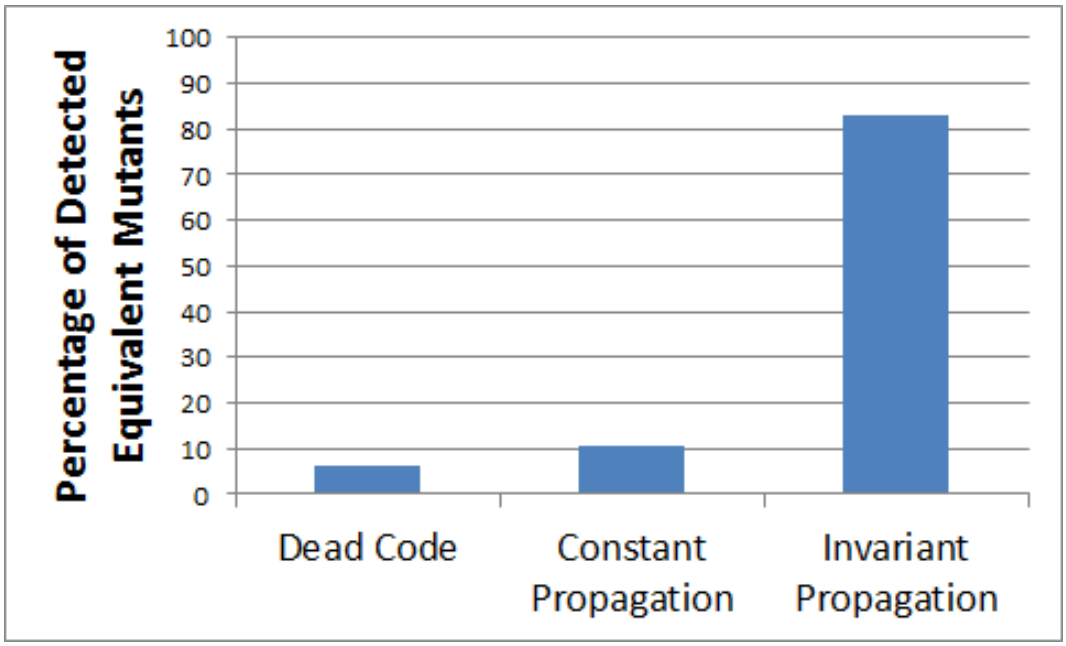


Figure 2.2: Equivalent Mutants Detected by Compiler Optimisation [116]

2.2.1.3 Detecting Equivalent Mutants Indirectly

Rather than prove the equivalence of a mutant, it may be possible to predict this based upon some other characteristics of its execution. The higher the impact (for some measure of impact) that a mutant has on the program's execution, the less likely it is to be equivalent [57]. For example, Schuler and Zeller [140] suggested measuring impact as the difference made in the number of statements exercised and the values returned by each procedure.

Mutations that have an impact on the program's execution, as measured by Schuler and Zeller [140], were shown to have a 58-79% likelihood of being non-equivalent. Although this is only a slight improvement on the 54% they reported for all mutants, a better result can be achieved by only selecting mutants with the highest impact (see Table 2.2). Statement coverage appears to be the best indicator of non-equivalence - 93% of the top quarter of mutants that have an impact on statement coverage were shown to be non-equivalent.

Table 2.2: Detecting Equivalent Mutants Indirectly, adapted from [140]

Impact Metric	**Top 15%**	**Top 20%**	**Top 25%**
Coverage Impact	88%	91%	93%
Data Impact	88%	91%	86%
Combined Impact	90%	85%	76%

Mutants can also be distinguished by their memory usage or execution time [46]. The task of accurately measuring the resources used for the execution of a program is complicated by unpredictable factors such as cache usage and other interruptions. As a consequence of this, Ellims et al. [46] struggled to implement an effective tool for distinguishing programs by their running profile. It may be possible to simulate the execution of mutants in a more controlled way, but the author is not aware of research in this area. Indirect techniques may be able to predict equivalence when it is difficult to show it conclusively by other means. As a result, it may be possible to find more equivalent mutants using these techniques. However, the end result is only a prediction and manual inspection is often necessary to confirm whether mutants really are equivalent.

2.2.1.4 Preventing Equivalent Mutants

It can be computationally expensive to identify mutants as equivalent. Therefore, it may be useful to prevent equivalent mutants from being produced. One way to achieve this is by careful selection of the mutation operators. Some operators create a higher percentage of equivalent mutants than others (see Figure 2.3). Mresa and Bottaci [109] took this into account when deciding which operators to use. Their experiments suggest five operators as the most efficient in terms of their ability to produce useful test data and avoid equivalent mutants.

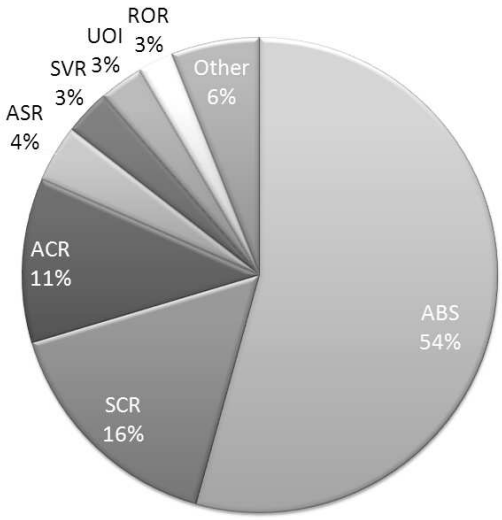


Figure 2.3: Proportion of Equivalent Mutants by Operator [116]

If no test data can be found to detect a mutant, the mutant is likely to be equivalent. Adamopoulos et al. [2] co-evolved mutants and test cases to make use of this intuition. Each test case is given a fitness value proportional to its ability to detect mutations and each mutant is given a fitness value proportional to its ability to avoid detection. Mutants that cannot be detected by any of the test cases are assigned a penalty value. This method is simple and fast, but it is also likely to eliminate mutants that are non-equivalent. Typically, more than enough mutants are produced to counteract this problem, but it may still be difficult to select which mutants are the most useful or interesting.

2.2.2 Too Many Mutants

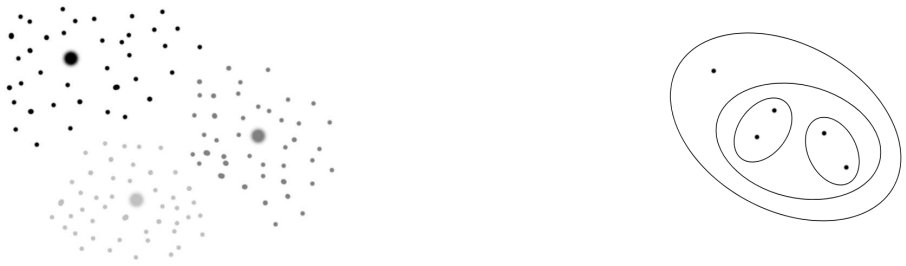
Budd et al. [24] showed that the number of mutants generated for a program is proportional to the product of the number of data objects and references. In most programs, this is approximately proportional to the square of the number of statements [1]. It is infeasible to evaluate every test case against all the mutants of a large program. Therefore, various techniques have been devised to reduce the number of mutants that are generated.

2.2.2.1 Mutant Sampling

The simplest reduction technique involves sampling mutants at random. It achieves a *cost reduction* that can be measured both in terms of the number of mutants that are sampled and also the reduction in the number of test cases necessary for them to be detected. Removing a mutant from evaluation may mean there are some faults that cannot be represented. Therefore, it is important that the analysis of test data is not significantly affected. An experiment by Mathur and Wong [156] showed mutants can be sampled to achieve a high cost reduction, without significantly affecting the mutation score of the generated test data (see Table 2.3). However, the blind nature of random sampling suggests a more intelligent approach might achieve even better results.

Table 2.3: Mutant Sampling [156]

	Mutation Score	**Cost Reduction**	
		in Mutants	**in Test Data**
10%-mutation	97.56%	90%	52%
15%-mutation	97.77%	85%	45%
20%-mutation	97.95%	80%	39%
25%-mutation	98.68%	75%	41%
30%-mutation	99.10%	70%	33%
35%-mutation	99.26%	65%	33%
40%-mutation	99.39%	60%	31%



K-means Agglomerative

Figure 2.4: Two Clustering Techniques

2.2.2.2 Mutant Clustering

One 'smarter' approach towards mutant selection divides the mutants into groups (or clusters) with *K-means* or *Agglomerative* clustering algorithms (see Figure 2.4). K-means clustering starts with k empty clusters and then adds mutants one at a time. Agglomerative clustering first treats every mutant as a separate cluster and then iteratively merges them together. Unlike random sampling, these techniques guarantee a degree of variety in the mutants that are selected.

Hussain [74] measured the similarity of mutants as the number of bits by which they differ. This is known as their *hamming* distance. Although this measure is effective in the experimental results of Table 2.4, it may be too naïve for operators that add or remove statements from a program. The results are also difficult to compare with random sampling because they do not take into account the computational effort required to divide the mutants into clusters.

Table 2.4: Mutant Clustering [74]

	Mutation Score	**Cost Reduction**	
		in Mutants	**in Test Data**
K-means	99.64%	86.90%	54.72%
Agglomerative	99.12%	86.69%	55.74%

2.2.2.3 Selective Mutation

Rather than sampling or clustering mutants post-priori, it can be more efficient to prevent them from being generated. This can be achieved by removing members of the operator set that are determined to be less efficient. In experiments with a mutation tool called Mothra (see Appendix A.2), two of the mutation operators (ASR and SVR) together produced 30-40% of the total number of mutants [120]. Test data sufficient to kill the mutants produced by these operators, were also able to kill more than 99.99% of the other mutants as well (see Table 2.5).

Two-selective mutation only reduced the number of mutants by 24%, but removing four and six of the operators removed 41% and 60% of the mutants respectively (see Table 2.5). Although none of these techniques reduced the number of mutants as much as clustering, selective mutation does not carry the cost of dividing the mutants into groups because it only generates mutants that are used. Another approach reduced the number of mutants by 80%, using just two operators (ABS and ROR) to achieve a mutation score of 95% [156].

Table 2.5: Selective Mutation, based on [120]

	Mutation Score	**Cost Reduction (in mutants)**
2-selective	99.99%	24%
4-selective	99.84%	41%
6-selective	99.71%	60%

Offutt et al. [118] divided mutation operators into three classes according to whether they replace operands, modify expressions or change entire statements. Expression-modifying operators were found to be the most efficient, achieving a high mutation score and cost reduction (see Figure 2.5). Offutt et al. [118] suggests this is because most statements in a program contain expressions that can be mutated. Expression mutation modifies every variable (except those on the left-hand-side) and every constant (except True and False). It therefore offers high statement coverage with little cost. Yet, as with selective mutation, the results do not show how many test cases are needed to kill the mutants produced.

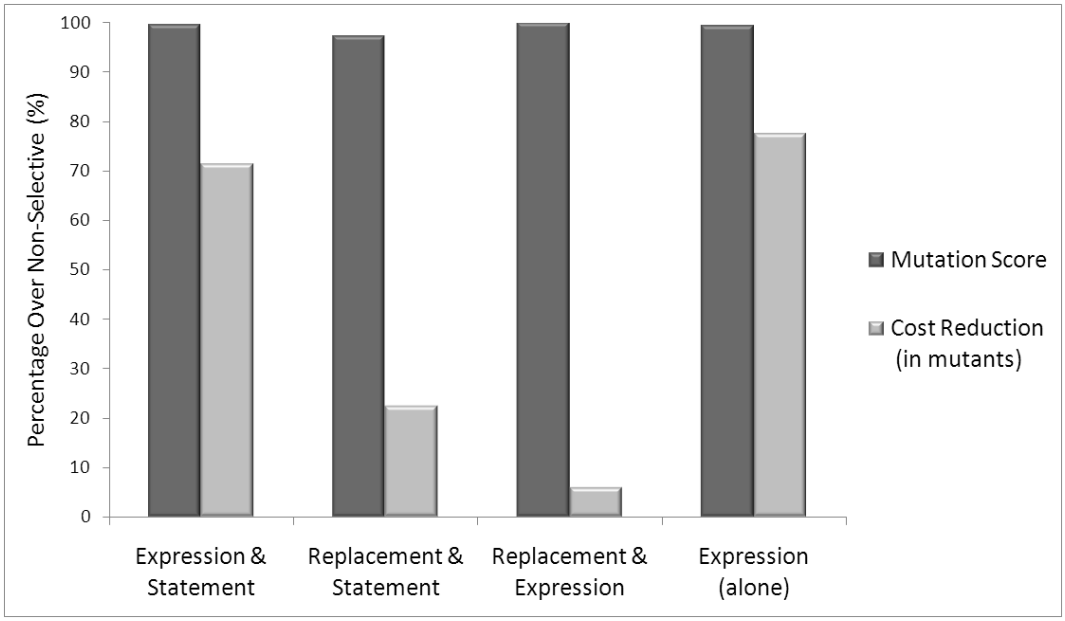


Figure 2.5: Class-based Mutation, based on [118]

It would require an exhaustive search to find the most efficient selection of operators. Yet statistical approximations can be made using techniques, such as least angles regression (LARS). LARS increases the coefficient of the operator with highest correlation to the complete set (see Figure 2.6). The coefficient of x_1 is increased from s_0 to s_1, then at s_1 both coefficients are increased together. Namin et al. [112] applied LARS to select 28 operators out of a total 108. These operators only produce 8% of the total number of mutants, yet Namin et al. [112] claim they are sufficient to accurately predict the test data's performance.

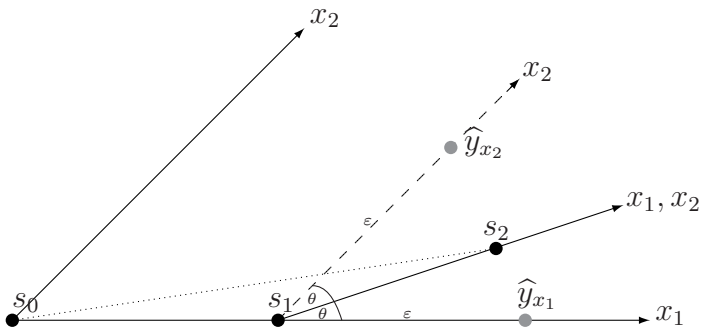


Figure 2.6: Least Angles Regression (LARS)

Much research has been conducted into choosing a small set of operators that maximise the mutation score of test data produced from them. However, for mutation analysis to be rigorous, it is important to include a sufficiently wide variety of mutants. Budd et al. [24] suggested that certain operators may be more efficient for representing different kinds of faults. Barbosa et al. [12] described a procedure for selecting a small set of operators that not only maximises the mutation score that can be achieved from them, but also ensures that at least one operator from each class is used. Figure 2.6 shows the results of an experiment involving 27 programs. Barbosa's sufficient procedure was shown to perform better than both expression mutation and 6-selective mutation.

Table 2.6: Various Techniques for Selective Mutation [12]

	Mutation Score	**Cost Reduction (in mutants)**
Expression Mutation	0.97	78.1%
6-Selective Mutation	0.992	47.9%
Sufficient Mutation	0.997	65.0%

Summary of Selection Techniques It is difficult to make definite conclusions about the effectiveness of mutant reduction techniques based on the small programs typically used in research into mutation analysis. For example, most of the programs used in the research of Barbosa et al. were under 30 lines of code. There are also issues in comparing the results of techniques for mutant reduction, because they were evaluated for different test programs, some in the C programming language and others in FORTRAN. Some evaluations measure cost reduction in terms of the number of mutants and others in terms of the number of test cases, when really neither of these tell the whole story with regard to human and computational cost. A thorough evaluation of mutant reduction techniques is needed to determine their adequacy and efficiency for programs of the kind that are currently used in industry. Only then can a conclusion be made as to which technique is the most effective.

2.2.3 Unrealistic Mutants

Simple syntactic mutants can be unrealistic as they are often easy to detect by a competent programmer. Complex faults may arise from the compound effect of multiple simple faults, or the occurrence of a fault in the semantics of a program. It is not always possible to correct complex faults with a small mutation in syntax. Therefore, in this section we consider some alternatives.

2.2.3.1 Higher-Order Mutants

Higher-order mutants are composed of a number of simple (first-order) mutations (see Figure 2.7). Higher-order mutants are a natural way to represent complex faults that occur from the compound effect of simple faults in a program. The coupling effect suggests that test data should be capable of detecting most of the higher-order mutants if it can detect all of the first-order mutants. Offutt [115] demonstrated for a program with 521,731 second-order mutants, that only 46 of them could not be detected by a test set adequate for all the first order mutants. It has also been claimed that higher-order mutants are able to represent more realistic faults than first-order mutants [64]. Therefore, it might be worth considering some of the higher-order mutants that are not coupled to their first-order components to achieve a thorough analysis of a program through mutation.

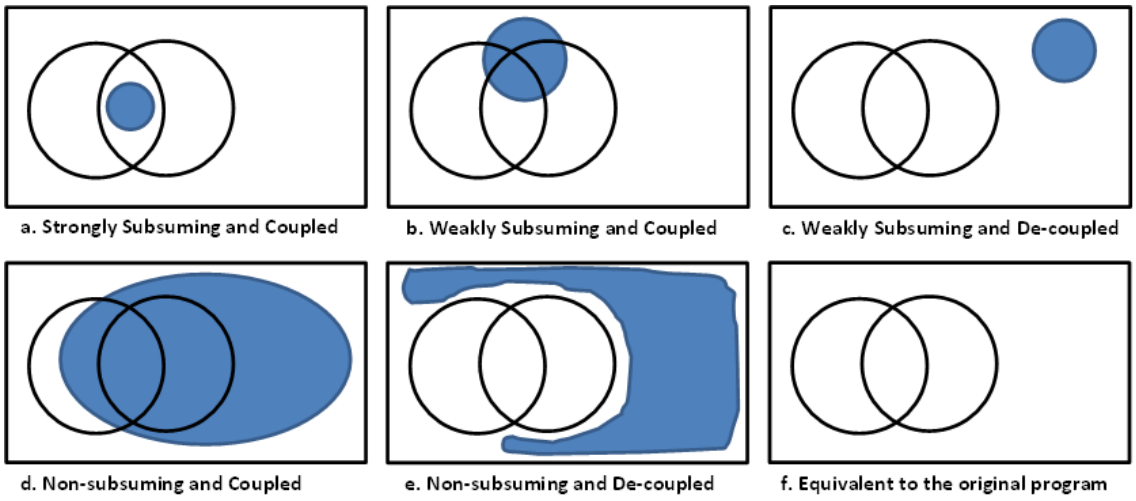


Figure 2.7: Classification of Higher-Order Mutants [64]

Jia and Harman [79] suggest classifying higher-order mutants according to whether they are coupled and/or subsuming. Some examples of this classification are shown in Figure 2.7. Higher-order mutants (c) and (e) are *de-coupled* because they cannot be detected by test data used for any of the first-order mutants they are composed of. Mutant (c) is particularly hard to detect because it is also *subsuming*, i.e. it affects output for a smaller proportion of the input domain than either of its first-order mutants. The first-order mutants in (f) interact in such a way as to cancel out their effect on the output, forming a (equivalent) higher-order mutant that cannot be detected with any test data. Jia and Harman [79] focus their attention on mutants such as (a) that are strongly subsuming.

A higher-order mutant is strongly subsuming if any test case that is able to detect it can also detect all the first-order mutants of which it is composed. Around 15% of subsuming mutants are strongly subsuming [64]. Although higher-order mutants are often harder to kill, some of the computational expense associated with test data generation may be reduced by considering strongly subsuming mutants [79]. Test data produced to detect these mutants can detect the first-order mutants with fewer test cases. This should reduce the computational expense of test data generation, as there are fewer mutants to evaluate the test data against. Therefore, some higher-order mutants are particularly useful.

2.2.3.2 Semantic Mutants

Semantic mutants represent the result of programmer misconceptions over the way their software should be interpreted. Syntactic faults have a specific point of failure and propagate this effect towards the output. In contrast, a consistent misconception may become effective at many different lines of programming code, but only involve a single semantic fault. This challenges the competent programmer hypothesis because many small syntactic mutants might be necessary to represent one semantic fault. In this way, a small syntactic mutant can have a large effect on the semantics of a program and small semantic mutant may have a large effect on the interpretation of its syntax [117].

Semantic faults occur from misunderstandings of the specification, data context or programming language. Misunderstandings in data context can be represented by small syntactic changes to the program. For example, a reference

to an array can be made out of bounds simply by changing its subscript [48]. Similarly, specification models can be mutated by adding, deleting, modifying or negating conditions on the transitions between its states [5]. Misunderstandings of a program language can be represented by mutating the language specification or making syntactic changes to the program to simulate semantic faults.

Semantic mutants target areas of ambiguity or potential for confusion. The C language specification allows compilers to interpret a program in slightly different ways [68]. If a programmer becomes accustomed to one implementation, they may find that another implementation works differently to how they expect. The GNU Compiler Collection (GCC) has specific options for each of the different platforms it supports [142]. Mutation analysis can be performed by evaluating a program with a variety of compiler options. This indicates the adequacy of test data for misconceptions about the particular implementation of a language.

Even well-defined behaviour can still cause confusion. For example, a pointer to the first element of an array is treated as if it is the array itself within the file in which it is created, but it is not possible to reference the array this way from another file [68]. For these mutations it is necessary to rewrite the compiler. Alternatively, tools such as Bison translate compiler rules from Backus-Naur Form (BNF), which is conceptually easier to understand. Rule reconfiguration requires less effort, but rebuilding the compiler is more powerful.

An alternative representation of a semantic mutation is as a set of syntactic mutations. This offers finer grained control, as it is possible to apply a mutation to one part of the program without affecting every other part. This may seem less intuitive and more awkward, but it is more powerful than reconfiguring the compilation options and less computationally expensive than rebuilding the compiler. Clark et al. [34] suggest using a language called TXL to describe the transformation of a program under mutation. It takes a description of the structures to transform and the pattern/replacement pairs for mutation and transforms code into mutated code that represents a semantic change. Thus, semantic mutation can be represented in an entirely syntactic way.

2.2.4 Difficult to Kill Mutants

Some mutants are more difficult to kill than others. It is tempting to ignore mutants if they are found to be difficult to kill - after all, they are more likely to be equivalent to the original program. Yet, the competent programmer hypothesis suggests that difficult to kill mutants are more realistic. It is therefore important to generate test data for them. Two main techniques have been used to automatically generate test data to kill mutants: symbolic execution and search.

2.2.4.1 Symbolic Execution

Symbolic execution uses symbolic expressions and path constraints to predict the execution of a program. Rather than executing a program with actual input values, symbolic execution creates a symbolic value for each input parameter. Symbolic expressions are produced whenever arithmetic operations are applied to symbolic values and path constraints are used to describe the conditions under which a particular path will be exercised. One of the first applications of symbolic execution [35][82] was to generate test data to cover particular elements in the structure of a program by solving sets of path constraints (see Figure 2.8).

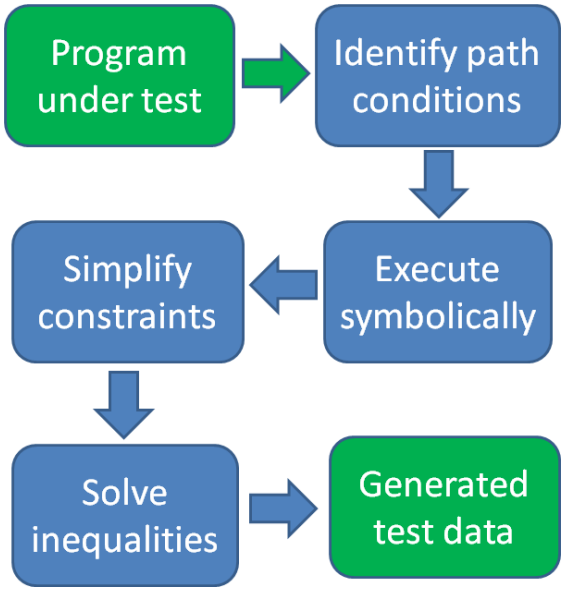


Figure 2.8: Using Symbolic Execution to Generate Test Data, adapted from [35]

No method is guaranteed to find solutions for every possible system of constraints [40]. However, most of the inequalities that arise from symbolic execution are linear. Linear inequalities do not contain products, powers or functions of variables. In other words, they take the form $a_1x_1 + a_2x_2 + ... + a_nx_n < c$. A number of techniques exist that can be used to solve systems of linear equalities efficiently in polynomial time [106]. These techniques may be augmented with a local search to solve simple non-linear inequalities [22]. Yet if there are no restrictions on the constraints, random search appears to be the most efficient technique [135].

The first attempt at constraint-based test data generation for mutation adequacy was made in 1991 by De Millo and Offutt [42]. Their approach collects path conditions for each mutant such that they hold true if the mutated code is executed. Further constraints are added that guarantee a change in the program's data if the mutation is exercised. No constraints are included to ensure this change reaches the output, as such constraints were believed to be too complicated [42]. This procedure was able to detect at least 95% of the non-equivalent mutants for five small Fortran programs, the largest of which had 55 lines of code.

De Millo and Offutt [42] used a constraint solving approach known as dynamic domain reduction. Each input parameter is given a large domain of potential values, which are then reduced by eliminating values that do not result in the desired branch being taken. The resulting subdomains can be used to execute a desired path through the program. If however, the resulting subdomains are empty, the path may be infeasible or the constraints are so complicated it is difficult to find a path. Back-tracking is then applied to try different alternatives.

Constraint solvers struggle with programs containing pointers. Mock data objects can be used to lessen this limitation, but they significantly reduce the precision of symbolic execution [146]. Floating point numbers can also make constraints difficult to solve, because they can be assigned a far greater range of values than integers. They can be approximated using intervals, which are propagated through the program to reduce their range for each constraint, but again this loses some precision [19]. These solutions have not yet been implemented in popular constraint solvers [91]. They are practical in nature, although they do not completely solve the problems with complex constraints.

2.2.4.2 Dynamic Symbolic Execution

It is possible to accurately evaluate the effect of any test input value by dynamically executing the program under test with the actual value, but it is not computationally feasible to do this for every possible input value. Symbolic execution can predict how software will behave for a wide range of inputs, but is not well suited to the evaluation of specific input values because it struggles with elements such as loop invariants and pointers. Dynamic and symbolic execution may be incorporated together to overcome some of the problems of both these techniques. The new hybrid technique is known as dynamic symbolic execution.

Larson and Austin [92] combine dynamic and symbolic execution by using symbolic representations of paths constructed during their dynamic execution to find more input values that allow the same path to be followed. In contrast, Godefroid et al. [54] use dynamic symbolic execution to find input values that result in alternative paths being taken. Both approaches combine the advantages of dynamic and symbolic execution but Godefroid et al. improve path coverage, whereas Larson and Austin improve coverage of the input domain.

Dynamic symbolic execution can be used to generate test data for programs that are too complicated to be handled by symbolic execution alone [25]. Symbolic and dynamic execution is applied in parallel. An alternative path through the program may be executed by negating one of the symbolic constraints and applying a automatic constraint satisfier to solve the new set of constraints, using dynamic values where helpful. Dynamic values may be used to help overcome the difficulties that symbolic execution has with pointers and non-linear constraints. As the constraints generated by dynamic symbolic execution are simpler, the constraint satisfier may be applied more efficiently.

Recently, dynamic symbolic execution has been applied to automatically generate test data to kill mutants [159] [128] [65]. Zhang et al. [159] used the Microsoft Pex framework to generate test data for programs written in C#. Papadakis and Malevris [128] used their own framework for programs written in Java. Harman et al. [65] also applied dynamic symbolic execution for Java, but used higher order mutants. These tools were able to generate effective test data for relatively large programs, the largest of which (Space) has 9,564 lines of code.

Zhang et al. [159] generate test data to weakly kill mutants. Rather than mutating the program under test, they instrument its code to introduce constraints that ensure a change in data. This is achieved by inserting a branch for each mutation (see Figure 2.9). If the branch holds true, a (localised) change in data will occur. Weak mutation killing does not ensure this change is propagated through to the output. Therefore, Zhang et al. [159] generate a number of test cases for each mutant in the hope that at least one of them will reveal a difference. Zhang et al.s approach killed all the non-equivalent mutants from a 98 line C# program.

Papadakis and Malevris [128] also use branch constraints to represent the conditions needed for a change in data. However, unlike Zhang et al., they include a technique to encourage these changes to propagate to the output. Papadakis and Malevris [128] assume that the output is more likely to change if the mutant has an effect on the control flow. They therefore try negating the symbolic path conditions one at a time from the point of mutation to the output. Papadakis and Malevris' approach killed 520 out of 937 mutants for a 500 line Java program.

Harman et al. [65] apply dynamic symbolic execution and search-based software testing to kill first and higher order mutants. They attempt to cause the generated test data to propagate the effect of each mutant to the output by searching for branch conditions that maximally disrupt the execution path after the point of mutation. This is measured in terms of the number of branches visited that are different in the mutant to the original program. Once this is found, dynamic symbolic execution is used to target that path. They achieve an average 62% mutation score across 17 Java programs of various sizes.

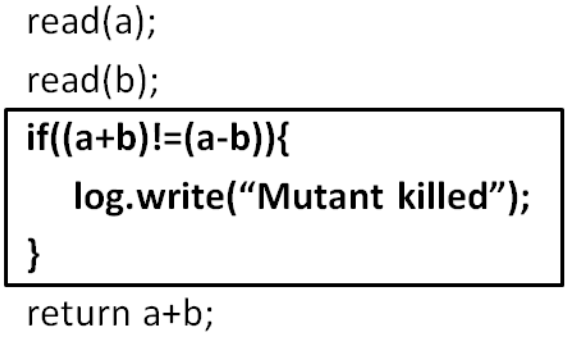


Figure 2.9: Instrumenting Code for Weak Mutation [159]

2.2.4.3 Search-Based Test Data Generation

In search-based test data generation, it is not necessary to specify how to produce an effective test suite. Search-based test data generation techniques use an evaluation criterion to search for optimal solutions. It is possible to use the mutation score of a test suite as the optimisation criterion, with the aim of maximising it as far as possible. Baudry et al. [13] implemented this criterion in both a genetic algorithm (GA) and a bacteriological algorithm (BA). They reported that the BA gave stable results with 95% mutation score, whereas the results of the GA were unstable with only 80% mutation score.

May [99] showed that an artificial immune system (AIS) is also superior to a GA when using mutation score as the evaluation criterion, as it is able to achieve a higher mutation score with significantly fewer program executions. May and Baudry agree the poor performance of the GA is likely due to it not remembering previous test cases. Both the AIS and the BA memorise a test case if it is able to reveal new mutants, allowing test cases to remain in the test data if they are useful even if they have a low mutation score.

Bottaci [20] presented a more sophisticated criterion based on whether the test data is able to reach each point of mutation in the programming code, and if so whether its effect is propagated through to the output. Bottaci's method requires that a path be identified from the input to each point of mutation. *Reachability* is measured as the extent to which each predicate in the path is not satisfied. For example, the predicate *a=b* would add *min(abs(a-b),L)* to the measure, where L is a large constant used to prevent overflow. Another metric, called *satisfiability*, is needed to measure how far the test data is away from causing a difference at the point of mutation. Bottaci proposed measuring *propagation* in terms of the number of unequal state pairs that occur between a mutant and the original program after the mutation point has been reached. Although Bottaci only proposed this heuristic conceptually, the reachability and satisfiability metrics have been implemented by Ayari et al. [8] in an ant colony optimisation algorithm, achieving a mutation score of 89% on the Triangle program.

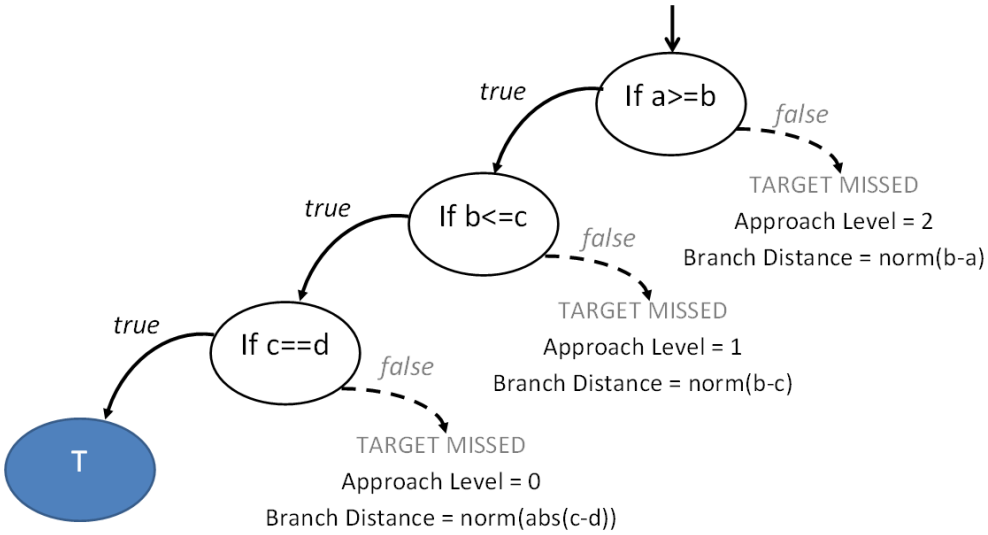


Figure 2.10: Example of Approach Level and Branch Distance [104]

Wegener et al. [150] also proposed a heuristic for reaching test data generation goals. Their approximation level criterion can be used to indicate the distance from the current path to the point of mutation. The first statement in the program has zero approximation level. From that point, the approximation level is incremented every time a critical branch is encountered that if taken, would prevent the goal from being reached. This means that statements with the highest approximation level are closest to the point of mutation (see Figure 2.10).

It is tempting, when targeting a particular goal for test data generation, only to focus on the critical branches that lead directly to that goal. Yet, it is also important to take semi-critical branches into account (see Figure 2.11). A semi-critical branch can only lead to the goal via a loop back to the statement from which it originated. This may seem like a backward step, but a certain number of loop iterations may be necessary before the next statement can be reached.

The previously described techniques for search-based software testing only take into account control and not data flow through a program. It is possible to take data flow into account by considering the statements that depend upon each other for their data. The *chaining* approach considers statements that must be executed based on the variables that they define [47]. For example, in Figure 2.12, the branch (8) that determines whether the goal (9) is reached is dependent on data defined in two statements (2 and 6). The chaining approach views these as

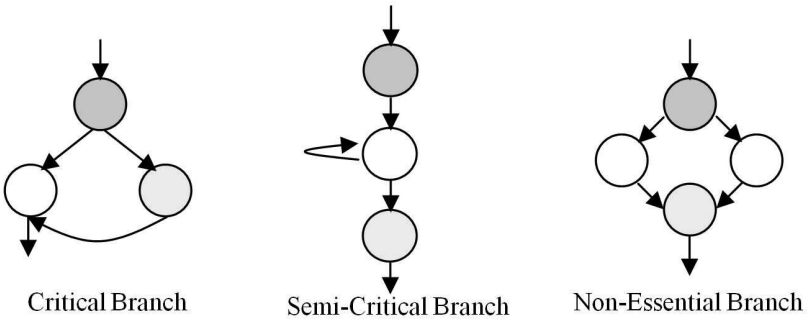


Figure 2.11: Goal Coverage: Three Types of Branch [84]

sub-goals, as one of them must be reached before the main goal can be achieved. Direct dependencies cannot reveal all the necessary data flow through a program. Therefore, indirect dependencies are also included in a data dependence graph. However, it can be too expensive to explore every path in the graph that lead to the goal, especially if the program contains loops. This may be one of the reasons this technique has not yet been used to generate test data for mutation adequacy.

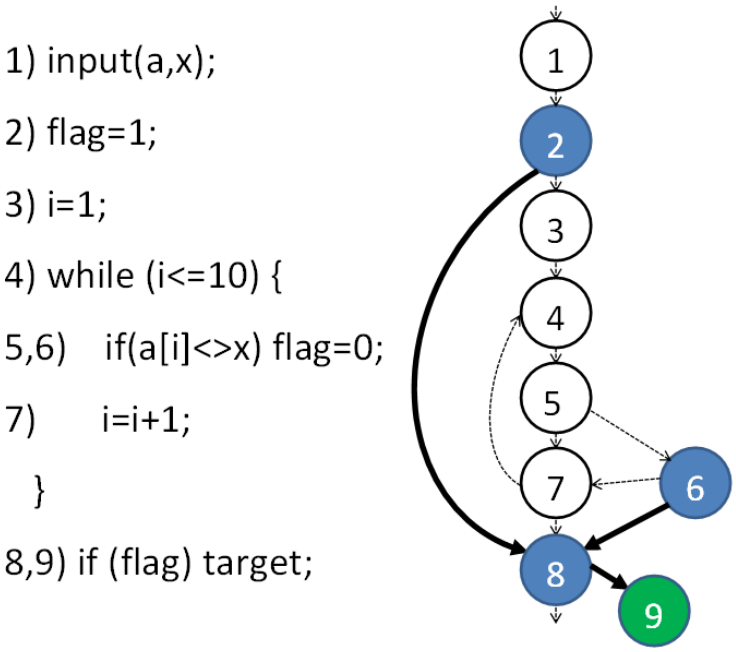


Figure 2.12: Example of the Chaining Approach [47]

Although evaluation criteria for search-based structural testing have been applied successfully many times to test generation [103], they have only been used in mutation analysis over the past few years. Fraser and Zeller [51] use a genetic algorithm to evolve test data capable of killing mutants. They measure how close a test suite is to exercising a mutant in terms of its approach level and branch distance. Branch distance is calculated as the difference between the current value of the expression in a branch condition and the value that is needed to exercise the intended branch [83]. Fraser and Zeller [51] measure the potential for a test case to propagate the effect of a mutant in terms of its impact on statement coverage and function return values (see Section 2.1.1.3).

More recently, Papadakis and Malevris [129] also guide search using the approach level, but include certain data dependencies so as to interpret it dynamically. They also incorporate Bottaci's satisfiability metric [20], as a measure of distance from effecting a change on the program's internal data state. Papadakis and Malevris [129] search for test data using the alternating variable method proposed by Korel [83]. The alternating variable method is a local search technique. Some researchers argue that local search techniques are inferior to population search (e.g. genetic algorithms) because they can get stuck in local optima [67]. However, local search can be restarted when necessary. Research by Harman and McMinn [66] indicates that local search techniques can be more effective for structural testing. Papadakis and Malevris [129] achieved an average 7.6% higher mutation score than the previously described work by Ayari et al. [8].

2.3 Improvements upon Random Testing

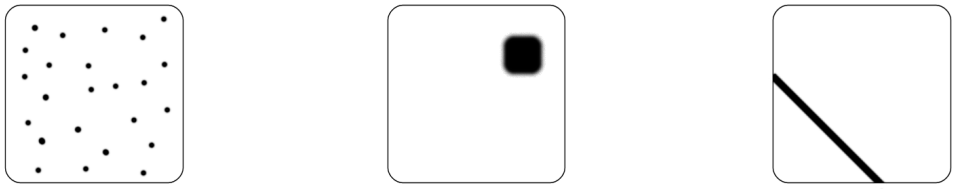
Random test data generation samples values evenly from a particular input distribution (often uniform). Design decisions are made regarding which distribution to use and how to select values from it. This thesis represents the first application of random test data generation to mutation analysis. One reason why random testing has not been used before now in mutation testing is that it can be inefficient at meeting testing goals (e.g. mutation coverage). Yet, even though random testing does not necessarily require any knowledge of the program code, such knowledge can be used to make random testing more efficient.

2.3.1 Random Testing

Random test data is typically straightforward and inexpensive to produce. It can represent any data type as a random bit string [76]. The biggest computational expense is in using an oracle to verify whether the outputs are correct [76]. It may be just as difficult to ensure the oracle works correctly as to find every fault in the software. Fortunately, some behaviour is clearly incorrect without the aid of an oracle. The York Extendible Testing Infrastructure (YETI) looks for assertion violations and runtime errors, such as invalid casting of values or division by zero [125]. In general, random testing is useful when there is not enough information in the software specification to accurately guide the choice of test data [59].

Random testing has been described as ineffective because it does not take into account the syntactic or semantic structure of a program [110]. However, it was able to achieve 93% decision coverage over five programs, with at most fifty test cases for each one [44]. Random testing is suitable for predicting a reliability bound because it can cover the entire input domain with test data [59]. It may have difficulty finding faults that occur for a small section of the input domain, so more targeted techniques typically can perform better test for test. Yet, the low overhead of random testing means it can reveal more faults per unit time [97].

Some guidance as to how to efficiently generate test cases is available even when little is known about the software under test. Chan [26] described three typical software fault patterns (see Figure 2.13). In the point-pattern, failure-causing inputs are not concentrated into regions by any particular characteristic, but rather scattered over the whole input domain. If this is the case, little can be done to reduce the number of test cases needed to find the first failure [31]. Frequently however, failures occur grouped together either in strips or in blocks. The *strip* pattern is typical when a domain error causes the input domain to shift so that values towards the edge of the correct range result in the program following the wrong path [154]. The *block* pattern is typical when the correct path is followed, but a computation error causes an assignment to result in incorrect output for a closely related set of input values. For these two patterns, the likelihood of finding the first failure may be improved by spreading the test cases as evenly as possible over the input domain [31].



Point Pattern Block Pattern Strip Pattern

Figure 2.13: Three Fault Patterns [26]

2.3.2 Adaptive Random Testing

Adaptive random testing (ART) uses the distance between test data to ensure an even distribution over the input domain. From a randomly generated set of test data, the one with the maximum distance from those that have already been executed will be chosen (see Figure 2.14). Chen [31] uses the Euclidean distance between integer input values, but it may also be possible to use other measures. The relationship between distance in the input domain and output range is unclear and might not be linear. Even so, their results show this technique can outperform traditional random testing by as much as 50%.

An early bias towards the corners and edges of the input domain may still result in an uneven distribution of test cases. This can be eliminated by measuring distance continuously, with the domain edges wrapped around to meet each other [101]. The domain edges, however may be the most interesting areas for test analysis [138]. It has also been claimed that it is only necessary to measure the distance to test cases that are nearest to the new candidate [30].

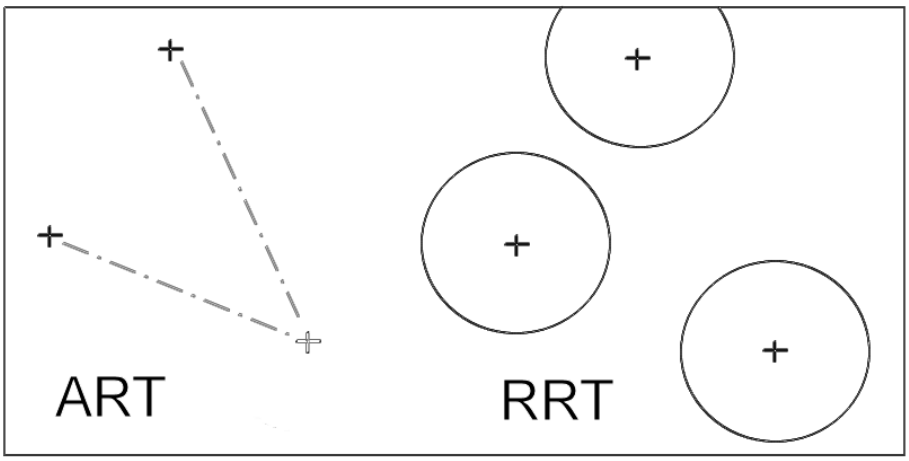


Figure 2.14: Adaptive and Restricted Random Testing

In contrast to ART, which adapts the order in which pre-generated test cases are selected, Restricted Random Testing (RRT) sets up exclusion zones around each non-failing input and randomly generates new test cases outside their boundaries (see Figure 2.14) [28]. Exclusion zones are decreased in size as more test cases are added to enable points closer to those already tested to be used. This should also help in maintaining the size of the excluded area. RRT is consistently able to improve upon random testing, with similar performance to ART [28].

The highest error detection rate is achieved when the exclusion area is close to its maximum, but if actual coverage approaches 100% the failure patterns may always be excluded. The ideal exclusion area is strongly influenced by characteristics in the input domain and is not easy to identify prior to testing. Chan [27] introduces normalised RRT to mould the excluded area to the input domain and provide more information regarding its maximisation. If the exclusion area is set appropriately, both the original and normalised RRT outperform ART.

2.3.3 Partition Testing

Partition testing offers another approach towards the distribution of test data. It divides the input domain into a number of subdomains and selects values from each one. Early experiments indicated little difference in performance between partition and random testing [44] [61]. Since then, Chen and Yu [32] have shown that it is possible to guarantee partition testing performs no worse than random testing, by selecting test data proportional to the size of each partition. Ntafos [114] criticises proportional selection, claiming it approximates random testing too closely. It can be more efficient to select test data from those partitions more likely to reveal a fault [60]. Whilst partition testing is classically a black-box technique, it performs better when something is known about the program code.

Partitions can be chosen from the specification or code structure. Ideally, they should be homogeneous, i.e. either all or none of their elements will cause a failure [153]. The only way to ensure homogeneity would be to place each value in a separate partition, so this has to be approximated instead. Code-based partitions measure homogeneity according to which structural components are exercised by the test data. It can be computationally expensive to counteract any imperfections, because this requires an analysis of the software's execution [60]. Specification-based partitions measure homogeneity according to which inputs are expected to give the same output. Combinatorial and boundary analysis can be used to compensate for imperfections without an analysis of execution.

Boundary analysis compensates for poor homogeneity by testing input values close to the point where one partition transitions into the next. This may reveal input values that should be in one partition according to the specification, but actually behave as if they are in another. One experiment showed the probability of finding the first failure to more than double when using boundary analysis [138]. Unfortunately, this almost doubled the amount of test data as well. More convincing results come from a comparison with random testing [138]. A similar fault-finding probability was achieved selecting one test value from each partition, as with the same number of random tests. However, over fifty-thousand random tests were required to match an average of just 25.1 values in boundary analysis.

Combinatorial analysis assumes software behaviour is dependent on the combination of value assignments to input parameters. It can cover all possible combinations with about a sixth to a half the number of test inputs as random testing [86]. Unlike random testing, it guarantees which combinations are tested and so is able to identify rare interactions between the input variables. A pairwise scheme can significantly reduce the number of test cases from that required for exhaustive testing, yet ensures every combination of values is applied for each pair of input variables [36]. For example, there are 27 ways of combining three values for each of three variables, but only 9 pair-wise tests are required. Some faults are caused by interactions involving more than two variables [88]. However, over 70% of program faults can be detected by pairwise testing [87] and 100% of faults can be detected with 4-6 way interaction [88]. Therefore a relatively small test set can be used to test a large number of variables. The biggest limitation of combinatorial testing is the need to choose a set of discrete input values. This may be difficult for applications with continuous input variables. However, if this is possible, the performance increase over random testing is significant.

2.3.4 Partition and Adaptive Random Testing

Partitions have been applied to adaptive random testing (ART) with little additional cost [30]. ART by random partition splits the input domain at the most recently executed test case and chooses the next input from the partition with the largest area. The previous test case will be at the edge of the new partition, so the next test case is likely to be chosen at some distance from the previous one. ART by bisection repeatedly splits the input domain into partitions of equal size and chooses the next input from a partition that does not contain any test cases. Each test case will be in its own partition, so should be evenly distributed over the input domain. Both schemes reduce the number of test cases needed to find the first failure by about 25% for the block pattern and 5% for the strip pattern. As expected, the improvement for the 'point' pattern is negligible.

Table 2.7: Proportion of test cases before fault found with ART Methods [100]

	Block	**Strip**	**Point**
Restricted Random Testing	0.6182	0.9226	0.9741
Adaptive Random Testing (ART)	0.6512	0.9244	0.9648
ART by random partition (RP)	0.785	0.972	0.9666
ART by bisection (Bi)	0.7362	0.955	0.9561
ART-RP with RRT localisation	0.6904	0.9538	0.9523
ART-RP with ART localisation	0.7194	0.9578	0.9588
ART-Bi with restriction	0.672	0.93	0.9252

Partition-based approaches to adaptive random testing are computationally inexpensive because there is no need to measure the distance between individual test cases. However, two test cases may be placed very close to each other if they are generated near the boundary separating adjacent partitions. As can be seen from Table 2.7, this increases the number of test cases that are necessary to find the first failure. Chen attempted to solve this problem by introducing *localisation*. ART by localisation is similar to ART by random partition, but applies restriction or distance based adaptive random testing within each partition [30].

The best results with partitioning can be achieved using ART by bisection and restriction, yet even this combination is not as effective as partitionless ART techniques [100]. Counter-intuitively, the partition-based techniques perform better than the non-partition techniques for point data. No reason was found why this might be the case. Localisation can be seen as a useful compromise between the low computational cost of just using partitions to distribute the test cases and the effectiveness of just using restriction or measures of distance.

2.3.5 Testing in High Dimensionality

It is challenging to create an even distribution of test data for software modules with many input variables. As dimensionality increases, test cases are distributed more sparsely and the relative distance between the nearest and furthest neighbours is negligible [15]. Small biases in a testing strategy are amplified in higher dimensions [89]: distance-based ART will develop a greater edge-bias and partition-based ART a greater centre-bias. A simple measurement of distance is not sufficient to guarantee an even distribution, because it does not take into account the need for test data to vary in every dimension.

Kuo et al [90] suggest filtering the test cases to ensure they are compared in terms of their input parameters as well as their distance from each other. A variable v is used to indicate which of the immediate neighbours are eligible for selection (see Figure 2.15). A second variable r is used to allow v to take a strict value at first, then gradually relax if insufficient neighbours can be found.

Chen et al. [29] measure how evenly the test cases are balanced around the centre of the input domain. Intuitively, if the test cases are well balanced they are also likely to be evenly distributed. The problem with this technique is that it results in a 'black hole effect', whereby all the test cases are pressed into the centre. To compensate for this, Chen et al. generate test cases within a small region in the centre of domain, then incrementally extend the domain.

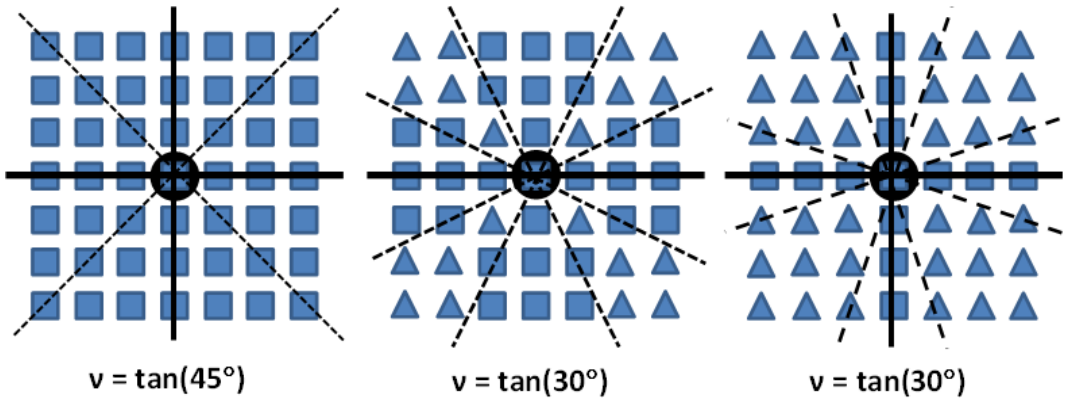


Figure 2.15: Restricted Neighbourhood for High Dimensions [90]

2.3.6 Statistical Testing

Even if it is possible to distribute test cases evenly over the input domain, there is no guarantee that all the structural components in the program code will be exercised. In fact, given a uniform input distribution, the frequency with which a particular component is exercised will be inversely proportional to the region within the input domain that can exercise it. This is because each branch condition in the program code limits the input values that are able to exercise the paths that it controls access to. Testing the structural components evenly typically requires a highly specialised input distribution.

Statistical testing manipulates the input distribution from which test cases are sampled so that each structural component has as near as possible the same probability of being exercised [145]. Every statistical testing strategy includes at least two parameters: the input distribution and the size of the test suite. The input distribution should be designed to increase the probability of exercising the elements that are otherwise least likely to be executed. An input distribution can be generated statically [55] or dynamically [145] to ensure a sufficiently high probability for the least exercised components. It is necessary for the test suite to be large enough to make sure that the least likely structural component has a high probability of being exercised at least a couple of times.

Whittaker and Thomason [155] present a Markov-chain model approach to statistical testing, whereby state transitions represent particular action that may be applied to a program (e.g. moving the cursor to a menu item or pressing the enter key). Their approach is best suited to testing interactive programs. Poulding and Clark [131] present an approach to adapting statistical testing for numerical programs. They model the dependencies between parameter values with a Bayesian network representation of the program under test. The input distribution is represented as a series of bins, the sizes of which affects the frequency with which particular input values are sampled. More recently, Poulding et al. [132] have used stochastic grammars and hill-climbing to develop effective models of the input distribution more efficiently.

2.4 Static Analysis

Static analysis evaluates a program directly from its source code rather than the results of an actual execution. It is able to identify incorrect behaviour without the need for any input data. Static analysis techniques fall into two main categories: code scans and abstract interpretation

2.4.1 Code Scans

The source code can be scanned to look for properties that suggest the presence of faults. For example, checks can be made as to whether a divisor will ever become zero. Static analysis can also detect undesirable properties, such as code that cannot be reached and loops that do not terminate. There are two main forms of code scanning technique: finite state automaton and data flow analysis.

Program statements are fed line-by-line into a purpose built finite state automaton (FSA) designed to reach an accepting state whenever a particular property is found. For example, an FSA could be used to check the program's behaviour when reading data from an input stream. If fewer bytes are returned than expected, the program might continue reading past the data into values that have not been initialised. Therefore, a warning should be triggered whenever data is used immediately after it is read, unless its actual availability is checked first [73].

Data flow analysis techniques are more sophisticated; they can be used to check for undesirable properties, such as input streams that are left open when they are no longer used. It is necessary to consider every path in which a stream is open because any path in which it is not closed may cause memory leaks and other problems [73]. Data flow analysis can be used to trigger a warning if the program terminates whilst one of the streams is still open.

Findbugs [73] uses FSAs and data flow analyses to look for commonly occurring bug patterns. It has not yet been used in mutation analysis, but mutants could be produced using the information it provides to target particular faults that are likely to occur. It might also be possible to use it to guide the search for test data that can distinguish mutants from the original program. Code might not find every fault in the program and some warnings may not correspond to actual faults, but they do give a prediction as to where faults are likely to occur.

2.4.2 Abstract Interpretation

Abstract interpretation makes it easier to find faults by simplifying the program's semantics and reducing the domains of its variables. For example, Figure 2.16 describes the use of abstract interpretation to identify conditions for a program's termination. The variable x is considered odd or even without regard to its actual value. If the program does terminate, x will be even at the output, regardless of its value at the input. If x is odd at the input, the program will definitely terminate, but if x is even, this cannot be guaranteed. Abstract interpretations lose some of the original information - actually, only an input value of zero would cause the program not to terminate. Yet, they still reveal details that might not otherwise be obvious. The most important characteristic of a valid abstract interpretation is that it will never claim a program is correct when it is faulty.

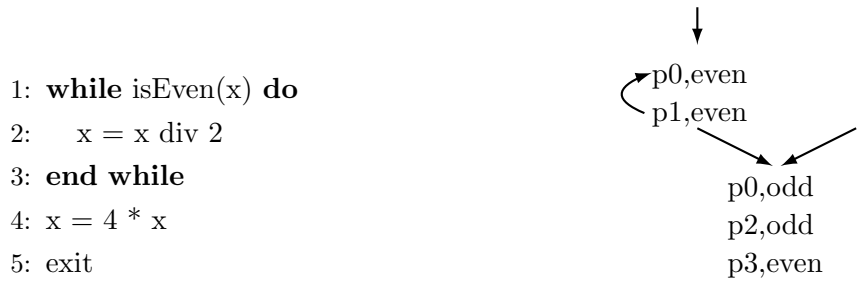


Figure 2.16: Example of Abstract Interpretation [139]

Cousot [37] describes how abstract interpretations can be *relational* or *non-relational* in nature. Relational abstractions are more precise, because they take into account potential relationships between variables. A non-relational abstraction might replace integers in a program with their sign (positive or negative), ignoring their absolute value. This simplifies analysis, but it loses any relational information. For example, it is impossible to encode that $x + y$ will always be below five in the sign (or any other) non-relational abstraction. In contrast, a relational abstraction would use this inequality along with others to produce a bounded area in which we know the values of x and y must lie. Abstract interpretations have to trade between expressive power and computational expense.

2.4.3 Summary

Mutation analysis is a powerful technique for evaluating the effectiveness of test data, but it faces a number of challenges: the computational cost of generating mutants and applying them to test cases makes mutation analysis more expensive than other testing techniques; equivalent mutants can have a significant effect on the precision of mutation testing but are difficult to remove by automated means. Mutation analysis can also be more labour-intensive than other techniques. A large number of test cases are required to kill all the mutants of a program and it requires human effort to identify what their expected output should be. This thesis addresses the human cost of mutation analysis by optimising subdomains from which test suites can be drawn that are efficient at killing mutants.

Random testing is typically less efficient than structural testing because it does not take into account any internal properties of the program under test. Techniques have been developed that reduce the number of test cases needed to find the first fault by spreading them evenly over the input domain, but there is a limit to how much of an improvement can be made without any knowledge as to how the program actually works. Rather than sampling test cases from the entire input domain, my approach uses subdomains that have been optimised to kill mutants. I use static analysis to identify the most difficult to kill mutants because test suites capable of killing these mutants are also likely to kill other mutants as a side effect. Ultimately, subdomain testing requires less human effort than other techniques because it selects test cases that are efficient at killing mutants and they are close together in the input domain.

Chapter 3

Evolving Subdomains for Mutation Adequacy

3.1 Introduction

This chapter introduces a new technique for optimising regions of the input domain (known as subdomains) that can be sampled at random to produce small but efficient test suites that achieve a high mutation score. Most test data generation techniques aimed at mutation adequacy use Dynamic Symbolic Execution (DSE) to target the branch conditions leading up to a mutation (see Section 2.4). DSE is able to exercise particular paths through a program efficiently, but provides little guidance as to whether a mutation will affect the output. The technique presented in this chapter restricts random testing to subdomains that are efficient at killing mutants. It can be used to provide insight into the internal workings of a program without inspecting its program code.

The choice of subdomains has a significant effect on the efficiency of random testing. For example, the TriTyp (also known as Triangle) program has three integer inputs (a, b and c) and its branches contain conditions such as $a{=}b{=}c$. Michael et al. [107] selected over 8000 test cases from the entire input domain, but exercised less than half of the program's branches. Duran [44] selected 25 test cases from the subdomains ([1,5], [1,5], [1,5]) and exercised all the branches. Random testing can be made more efficient by carefully tuning the subdomains.

It can be difficult to determine which subdomains to use and why. For example, Andrews et al. [7] report that the subdomain [0,31] gave the best results in testing a dictionary, but do not explain how they discovered this 'magic number'. Similarly it is difficult to determine in advance whether sampling from uniform, normal or exponential distributions will be more efficient. Rather than applying random testing through a process of trial and error, it is better to use a more systematic approach to determine the most efficient subdomains and sampling distributions for the program under test. This chapter introduces a technique for optimising subdomains to be used with a particular sampling distribution.

Subdomains are optimised using an evolution strategy (ES) so that test cases sampled from them are efficient at killing mutants. The process of evolving subdomains is computationally expensive, but once this is achieved, it is inexpensive to generate further highly efficient test suites. The technique is particularly suitable for regression testing because, although programs change with time, subdomains are likely to be evolved more quickly once suitable initial values have been found.

Subdomain evaluations can be used to provide a broad overview of program behaviour. They are particularly useful when direct source code analysis techniques cannot be used due to the size or complexity of the program under test. Like DSE, subdomains target mutants efficiently. However, unlike DSE they can also be used to generate further effective new test cases very quickly.

Subdomains have three main advantages over other testing techniques:

1. **They improve the effectiveness of random testing and provide a means to achieve a high mutation score automatically**

2. **They can be used as a starting point for regression testing more readily than a set of individual test cases (e.g. generated by DSE)**

3. **They provide information about the execution behaviour of a program that may be useful when constructing further tests**

The rest of this chapter is organised as follows. Section 3.2 describes my approach to subdomain optimisation in detail. Section 3.3 introduces some experiments to assess the effectiveness of subdomain optimisation and Section 3.4 presents their results. Section 3.5 summarises the significance of these findings.

3.2 Subdomain Optimisation

I optimise input subdomains using an evolution strategy. Subdomains are evolved, one for each input parameter so that test data can be sampled within the bounds of each subdomain. Evolution strategies optimise numerical values, but input parameters can have many different data types. It is therefore necessary to represent types such as strings and Booleans with numerical values so that they can be optimised. In this chapter, a candidate solution consists of a set of subdomains with intervals in the following three forms:

Numerical subdomains
: are represented with lower and upper boundary values (rounded to whole numbers in order to keep the subdomains simple). Values are sampled inclusively, such that the subdomain [3,6] also includes the values 3 and 6.

Boolean chance values
: are described with an integer value between 0 and 100. Rather than defining a boundary within which test inputs are sampled, this value represents the percentage chance that a particular test input value is 'true'.

Character array subdomains
: are fixed in length (by default to five characters). Each character is treated as a numerical subdomain and the selected values are then mapped to letters from a particular alphabet (e.g. the basic Latin alphabet).

Boolean chance values and character boundary values must be kept within a specific region of validity: Chance values must be between 0 and 100; character boundaries must be inside the alphabet. I avoided using modulo arithmetic to map values into a valid region because it affects the direction of an adaptation (i.e. increasing a chance value could increase or decrease the likelihood of a sampled value being true). Instead, I chose to replace chance values or boundaries that are generated outside the range of validity with the nearest valid value. This preserves the directionality of the adaptation step, but does not preserve its magnitude. For example, values on the edge of the validity region have 50% chance of staying the same. Magnitude can be preserved by resampling until a valid value is found (see Algorithm 3), but I chose not to do this here for reasons of efficiency.

Evolution strategies are inspired by the process of adaptation in nature [10]. They were first used in the 1960s by Bienert, Rechenberg and Schwefel to optimise aerodynamics [10]. Evolution strategies differ from some genetic algorithms in that they optimise numerical values rather than bit strings and focus on mutation over recombination [151]. They are ideal for fine tuning numerical properties, as any disruption from recombination is largely avoided. Amongst other applications, evolution strategies have been used to optimise image compression [9], network design [113] and web crawling [80].

This chapter employs a traditional $(1 + 1)$ form of evolution strategy. This means that the evolution strategy maintains one candidate solution (set of subdomains) at a time. For every generation, a single new candidate solution is perturbed from the current one using a normal distribution. Each candidate solution is represented using a numerical set of values $(x_1 \dots x_n)$, as determined by the numerical coding of each subdomain type described previously. A new candidate solution $(x'_1 \dots x'_n)$ is generated such that $x'_1 = x_1 + \epsilon_1 \dots x'_n = x_n + \epsilon_n$, where $\epsilon_1 \dots \epsilon_n \in \mathcal{N}(0, \sigma^2)$. The variance of the normal distribution (σ^2) is adjusted throughout the optimisation process so as to maintain an optimum rate of convergence. At each generation, the new candidate replaces the current solution if it evaluates as being superior, otherwise it is discarded.

Normal distributions are suitable for generating new candidate solutions because they favour values close to the old ones, but still allows exploration of values further away. However, if the variance of the normal distribution is too small, the evolution strategy may never reach a global optimum; if it is too large, an optimum could be passed over without being detected. Rechenberg's one-fifth rule is used to adapt the variance for optimal effectiveness [14]. The ideal convergence rate is presumed to be achieved when one out of five new values perform better than their parents. Following advice from Schwefel [141], this ratio is maintained by applying Equation 3.1 once every ten generations.

$$\sigma' = \begin{cases} \sigma * 0.85, & \text{if } r < 2 \\ \sigma / 0.85, & \text{if } r > 2 \\ \sigma, & \text{if } r = 2 \end{cases} \tag{3.1}$$

(where c is the number of successful adaptations in 10 generations)

Algorithm 1 describes the process used to optimise numerical subdomains ($\alpha \ldots \Omega$) with lower (l) and upper (u) boundary values. A similar process can also be applied for Boolean and character values, since the evolution strategy treats each value independently. The evolution strategy is set up so that it has the ability to explore a wide range of potential candidate solutions quickly, then narrow its focus to exploit those subdomain values found to be the most efficient.

Algorithm 1 Synthesising an optimal solution $[\alpha_l, \alpha_u]$, $[\beta_l, \beta_u]$, $\ldots$, $[\Omega_l, \Omega_u]$

1: Select initial random values $(x_1 \ldots x_n)$ for α_l, α_u, β_l, β_u, $\ldots$, Ω_l and Ω_u.
2: Generate s test cases from $[x_1, x_2]$, $[x_3, x_4]$, $\ldots$, $[x_{n-1}, x_n]$.
3: Count the number of mutants m killed by the test cases
4: **repeat**
5: $r = 0$
6: **for** $i = 1 \rightarrow 10$ **do**
7: Sample new values from a normal distribution:
$x'_1 = x_1 + \epsilon_1$, $x'_2 = x_2 + \epsilon_2$, $x'_3 = x_3 + \epsilon_3$, $x'_4 = x_4 + \epsilon_4$,
$\ldots$, $x'_{n-1} = x_{n-1} + \epsilon_{n-1}$, $x'_n = x_n + \epsilon_n$ where $\epsilon_1 \ldots \epsilon_n \in \mathcal{N}(0, \sigma^2)$
8: Generate s test cases from $[x'_1, x'_2]$, $[x'_3, x'_4]$, $\ldots$, $[x'_{n-1}, x'_n]$.
9: Count the number of mutants m' killed by the test cases
10: **if** $m' > m$ **then**
11: $x_1 = x'_1$, $x_2 = x'_3$, $x_4 = x'_4$, $x_{n-1} = x'_{n-1}$, $x_n = x'_n$, $r = r + 1$.
12: **end if**
13: **end for**
14: **if** $r < 2$ **then** $\sigma^2 = \sigma^2 * 0.85$
15: **else if** $r > 2$ **then** $\sigma^2 = \sigma^2 / 0.85$
16: **until** generations>300 **and** mutation score no longer increases

Subdomain values are initially assigned uniformly at random, between 0 and 100 for numerical and Boolean input parameters and within the size of the alphabet for character array parameters. The normal distribution has an initial variance of 50. Although these initial values are somewhat arbitrary, they were found to be a good starting point for the programs under test. The evolution strategy can, as its search progresses, move outside of these initial boundaries.

3.3 Experiments

Experiments were set up to answer the following three research questions in regard to optimising subdomains for mutation testing:

RQ1: Are test suites sampled from an optimised set of subdomains more efficient at killing mutants than unoptimised random testing?
A test suite can be considered to be more efficient if it achieves a higher mutation score with the same number of test cases. I use mutation analysis to determine whether optimised sets of subdomains are more efficient than unoptimised random testing for test suites of 10, 100 and 1000 test cases.

RQ2: Are some shapes of input distribution (within the evolved subdomains) more efficient at killing mutants than others?
Distribution shapes emphasise different parts of each subdomain. Normal distributions emphasise central values, exponential distributions emphasise smaller values and uniform distributions emphasise evenly. I optimise subdomains for each shape so as to determine which one is the most effective.

RQ3: To what extent do the relationships between subdomains and their mutation scores reveal information about the program under test?
I investigate the relationship between subdomain values (e.g. upper and lower boundaries) and the mutation score they achieve. Once the relative characteristics of efficient and inefficient subdomains is known, it should be possible to produce new more effective subdomains in the future.

3.4 Methodology

3.4.1 Methodology for RQ1

RQ1 is designed to compare the efficiency of test suites generated from optimised subdomains against unoptimised random testing. If optimised subdomains perform more efficiently than random testing, the same standard of testing can be achieved with fewer test cases. For example, if 10 test cases can be sampled such that they kill as many mutants as 100 or 1000 test cases, this can be considered as a ten-fold or hundred-fold improvement in efficiency. Subdomain optimisation can be considered to be successful if, for a variety of programs, the optimised subdomains kill more mutants with fewer test cases than random testing.

Subdomain optimisation is evaluated using test case values that are drawn from a uniform input distribution restricted to the range of each subdomain. Algorithm 2 describes how random values sampled between 0 and 1 are scaled to subdomain lower and upper values. Each random value is multiplied by the range of its corresponding subdomain, then added to its lower value. If the lower value becomes larger than the upper value for a particular subdomain during the course of its optimisation, the two values are swapped around.

Algorithm 2 Restricted Uniform Random Sampling

```
1: if lower > upper then
2:     swap(lower, upper)
3: end if
4: range = upper − lower
5: return rand() ∗ range + lower
```

For every generation of the evolution strategy, I record the mutation scores achieved by each test suite sampled from the current candidate solution's set of subdomains. In order to present the results with as great an accuracy as possible, I average the mutation scores achieved for each program and each size of test suite over 100 trials. By definition, the largest possible standard deviation in mutation scores is 0.5. Therefore, by calculating confidence intervals from a t-distribution, it is possible to show that 100 trials are sufficient to achieve at least a 90% chance that the actual mean mutation score lies within 10% of the sampled average.

I compare the mutation scores achieved by test suites sampled from optimised subdomains against unoptimised random testing with a [0,100] numerical interval, 50% Boolean chance and the set of alphabetical character values. These values were chosen because test cases sampled from the entire input domain are very inefficient. The mutants killed by a random test suite vary from one sample to the next. Some mutants are killed frequently, whereas others are rarely killed. Therefore, instead of measuring the average number of mutants killed by random testing, I estimate the probability of killing each mutant individually. The results are compared to the expected mutation score for a random test suite of the same size. If subdomain optimisation is successful, the average mutation scores achieved should be greater than that expected by random testing.

The number of mutants expected to be killed by random test suites of 10, 100 and 1000 test cases was calculated for each program in Table 3.2 using a combination of experimentation and probability theory (see Table 3.1). In Equation 3.2, $e(s)$ is the expected number of mutants killed for a random test suite sampled with s test cases, where for each mutant m, K is the number of test cases that killed the mutant and N is the total number of test cases from a large test suite. A test suite of 100,000 random test cases was used because it is important that N is much larger than s for accurate results. Numerical input values were generated from the interval [0,100], Boolean values were given a 50% chance of being true and character arrays were generated from the entire set of character values.

Table 3.1: Expected Mutation Score for Random Test Suites

Program	**s=10**	**s=100**	**s=1000**
Power	0.963	0.994	1.00
TrashAndTakeOut	0.787	0.958	0.988
FourBalls	0.356	0.756	1.00
TCAS	0.0499	0.0569	0.0599
Cal	0.766	0.948	0.957
TriTyp	0.394	0.779	0.924
Schedule	0.236	0.840	0.853
Replace	0.209	0.321	0.329

$$e(s) = \sum_{m \in mutants} 1 - (1 - K/N)^s \tag{3.2}$$

3.4.2 Methodology for RQ2

RQ2 is designed to compare the effect of different input distribution shapes on mutant killing efficiency. Research into statistical testing [131] has shown that programs require different emphasis depending on their internal control and data flow. The programs used in this chapter (see Table 3.2) process a variety of forms of computation, from control programs to programs that perform string processing or numerical calculations. It seems sensible to expect each shape of input distribution to be better suited to some programs than others. However, the extent to which they improve the tailoring of subdomains is not yet known.

Test suites have so far been generated by sampling inputs uniformly across the range of each subdomain. Random numbers are primitively generated from a uniform distribution between zero and one. Uniform sampling is therefore a straightforward process of scaling random numbers to the range of each subdomain (see Algorithm 2). Other shapes of input distribution are less trivial to sample, but may produce more efficient test suites. The second research question investigates this possibility by evolving optimised subdomains for uniform, normal and exponential samples then comparing the mutation scores achieved.

Algorithm 3 Restricted Normal Random Sampling [21]

1: **if** $left > right$ **then**
2: $swap(left, right)$
3: **end if**
4: $range = right - left$
5: $mean = range/2 + left$
6: $variance = (range/4) * (range/4)$
7: **repeat**
8: **repeat**
9: $x = 2 * rand() - 1$
10: $y = 2 * rand() - 1$
11: $w = x * x + y * y$
12: **until** $w < 1$
13: $w = x * sqrt((-2.0 * ln(w))/w)$
14: **until** $abs(w) \leq range/(2 * variance)$
15: **return** $mean + variance * w$

Normal sampling is performed using the polar form of the Box-Muller transformation (see Algorithm 3). The polar form is computationally less expensive than the basic form as it does not require sine or cosine calculations. It constructs a circle such that $w = x^2 + y^2$, where x and y are random input variables, then uses sine and cosine rules to map them to a normal distribution. It is necessary to transform the values of x and y so that they are uniformly distributed between -1 and 1. The normal distribution is infinite, so as with exponential sampling some values must be discarded. Since 95% of the distribution probability area lies within 2 standard deviations either side of the mean, I set the standard deviation to a quarter of the desired range, so as to reduce the need for resampling.

Exponential sampling is performed using an inverse transform method (see Algorithm 4). In an exponential distribution, the probability that a random variable X is less than or equal to x (i.e. the cumulative distribution function) is given by $1 - e^{-\lambda x}$, where $x \geq 0$ and $\lambda > 0$. Rearranging this equation to make x the subject, $x = -ln(1 - P(X \leq x))/\lambda$. The exponential distribution is sampled by generating values uniformly at random between zero and one for $1 - P(X \leq x)$, then mapping them to the value of x that occurs with this probability. As exponential distributions are infinite, it is necessary to discard any values produced beyond the desired range and resample. In order to reduce the number of wasted samples, I set the value of λ so that 95% of the distribution probability area is inside the range. Rearranging the previous equation gives $\lambda = -ln(0.05)/range$, which is equivalent to $\lambda = ln(20)/range$.

Algorithm 4 Restricted Exponential Random Sampling

1: **if** $left > right$ **then**
2: $\quad swap(left, right)$
3: **end if**
4: $range = right - left$
5: $lambda = ln(20)/range$
6: **repeat**
7: $\quad x = -ln(rand())/lambda$
8: **until** $x \leq range$
9: **return** $left + x$

3.4.3 Methodology for RQ3

RQ3 is designed to determine whether it is possible to infer useful information about the programs under test from the relative efficiencies of subdomain values that have been evolved to kill their mutants. I address this research question by considering the mutation scores achieved by test suites sampled from subdomains with various sizes, lower and upper numerical boundaries, percentage chance values and centre points. The size of a subdomain is the difference between its lower and upper boundary values and the centre point is located half way between them. The aim is to discover whether there is a causal relationship between subdomain values and the efficiency of the resulting test suite.

I acknowledge there may not always be a direct relationship between the subdomain values evolved for a program and their ability to produce effective mutant-killing test suites. Take for example, a simple program that has one mutant ($M1$) and two input parameters ($I1$ and $I2$). Subdomains evolved for $I1$ are small and restricted to the same region of input, whereas subdomains evolved for $I2$ are large and spread over a wide area. It is clear that the subdomains evolved for $I1$ are important for killing $M1$. Subdomains evolved for $I2$ may be large because $M1$ can be killed by a wide range of values, or only a few values can kill $M1$ but they are spaced far apart. It is also possible that $I2$ has no affect on whether $M1$ is killed and the size is purely due to random drift. It is therefore necessary to consider why particular subdomain values have been evolved.

My analysis utilises every subdomain evaluation; not just the final selection of subdomains, but also those evaluated during the optimisation process. It is important to use all the evaluations so as to gain accurate information about the relationship between the mutation score achieved and the values of each subdomain. Using only the final selection would bias the results towards subdomains that achieve higher mutation scores and provide limited information about the distinction between high and low performance. I acquire information about the relationship between subdomains and their achieved mutation scores by visualising the results with scatter plots, histograms and surface area diagrams. Pearson correlation coefficients are also included where relevant. My intention is that, once the relative characteristics of efficient and inefficient subdomains is known, it should be possible to produce new subdomains more effectively in the future.

3.4.4 MuJava: Mutation Tool

I use MuJava to generate mutants for the programs under evaluation. MuJava includes twelve method-level operators and twenty nine class-level operators [95]. I only use the method level operators (see Appendix A.1), as the interaction between classes is outside the scope of this thesis. The method-level operators are based on research into selective mutation started by Wong and Mathur [156] and continued by Offutt et. al. [118]. Mutation operators that alter or replace expressions were found to be the most effective in terms of cost reduction and mutation score. MuJava method-level mutation operators modify arithmetic, relational, logical and conditional expressions in the program code [95].

3.4.5 Equivalent Mutants

Equivalent mutants have an adverse affect on mutation score because they cannot be killed by any test data. I therefore remove all the equivalent mutants before evaluating the results. This is achieved by executing mutants that are not killed at any point during the experiments with an additional one million random test input values. If none of the input values produces a difference, the mutant is checked manually for equivalence. Although no formal records were kept of the effort involved in determining the equivalency of mutants, in total I identified 356 equivalent mutants using this technique. This means that 9.8% of the mutants generated by MuJava from the programs under evaluation were equivalent to the original program. Most mutants were processed in less than a minute, but for some it took as long as 30 minutes to determine whether they were equivalent.

3.4.6 Random sampling

Random sampling is used to select the initial subdomain values and generate new candidate solutions for evaluation. I use the Mersenne Twister algorithm [98] because it is fast, has a long period ($2^{19937} - 1$) and passes all the Diehard tests for randomness. It must be initialised with a numerical seed as its starting point. Random seeds are important because poorly chosen seeds lead to repetitions within the size of the period. I use random seeds generated from atmospheric noise as a highly random source of data (see Appendix B.1) [58].

3.4.7 Test Subject Programs

The subdomain optimisation technique was applied to eight programs of varying size and complexity (see Table 3.2). The programs were chosen because they are frequently used in testing research (so are well known and understood), are written in Java (so can be mutated using MuJava) and are procedural rather than object-oriented. They range in size from 35 up to 500 lines of code and were mutated to produce between 58 and 1632 non-equivalent mutants each.

Table 3.2: Test Programs Used in the Experiments

Program	Mutants	LOC	Function	Source
Power	58	35	Calculates the value of x^y	[4]
TrashAndTakeOut	111	60	Mathematical calculation	[4]
FourBalls	189	40	Calculates the ratio of inputs	[147]
TCAS	267	120	Processes air traffic control	[128]
Cal	280	134	Counts days between dates	[4]
TriTyp	310	61	Classifies triangle shapes	[147]
Schedule	373	200	Determines execution order	[128]
Replace	1632	500	Performs substring replacement	[128]

Power, TrashAndTakeOut and Cal are straightforward programs that perform mathematical calculations. They were published in an introductory textbook on software testing [4]. **FourBalls** calculates the values of four integers (the weights of balls) relative to each other. It was originally used to evaluate an evolutionary test data generation technique [130]. **TCAS** is an air traffic collision avoidance system. It was first used by researchers at Siemens to investigate data flow and control flow coverage criteria [75]. **TriTyp** (also known as the triangle program) classifies triangles as equilateral, isoceles or scalene. It has been used extensively in test data generation research ever since it was introduced in early work by Ramamoorthy et al. [135]. **Schedule and Replace** are distinguished from the other programs by their use of character arrays. I represent the input file for Schedule as an array of instructions with specific command codes. For Replace, the search and replacement strings are limited to 5 characters and the source string to 10 characters (or copies of the search string). It was necessary to inspect the program code of Schedule and Replace to determine their input alphabets, but in practice this information may be available in documentation.

3.5 Results and Analysis

Below are the experimental results, addressing each research question in turn.

3.5.1 Results for RQ1

RQ1 is answered by optimising subdomains to produce test suites with different numbers of test cases, then comparing them against unoptimised random testing. Subdomains were evaluated with three sizes of test suite ($s = 10$, $s = 100$ and $s = 1000$). Subdomains are initialised randomly at the start of each experiment, then optimised with an evolution strategy (see Section 3.3). Table 3.3 compares the average mutation scores achieved for each size of test suites with those achieved by the initial random subdomains and those expected for the numerical interval [0,100], 50% Boolean chance and the entire set of character values.

Optimised subdomains achieve a higher mutation score than the initial random subdomains for every program with all sizes of test suite. The average optimised mutation scores also exceed the expected mutation scores on every program under test (see Table 3.3), with just three exceptions (shown in bold). In these three cases, all the mutants are expected to be killed by random testing, while the evolution strategy occasionally becomes stuck in a local optimum. In all other cases where random testing is not expected to kill all the mutants, the optimisation technique finds subdomains that kill more mutants.

Table 3.3: Difference Between Optimised Subdomains and Random Benchmarks

Program	**1) Compared to Initial**			**2) Compared to Expected**		
	s=10	*s=100*	*s=1000*	*s=10*	*s=100*	*s=1000*
Power	+5.04%	+5.72%	+6.30%	+2.45%	**-0.0645%**	**-0.0918%**
TrashAndTakeOut	+44.7%	+42.2%	+42.4%	+23.8%	+1.68%	+0.176%
FourBalls	+191%	+231%	+226%	+134%	+31.3%	**-0.733%**
TCAS	+379%	+377%	+376%	+533%	+728%	+687%
Cal	+125%	+127%	+130%	+15.9%	+0.859%	+2.00%
TriTyp	+126%	+90.4%	+83.2%	+62.4%	+22.5%	+33.0%
Schedule	+117%	+2.25%	+17.6%	+48.0%	+0.667%	+5.04%
Replace	+47.5%	+32.4%	+43.7%	+68.5%	+34.9%	+31.6%

Table 3.4 compares the effectiveness of subdomain optimisation with two experiments featuring dynamic symbolic execution (DSE). The average mutation score of the subdomains for TriTyp and Schedule outperformed that of DSE by a considerable margin. Yet with TCAS and Replace, even the highest mutation score achieved by the subdomains fell short of that achieved by DSE.

DSE is particularly effective for programs that have complex branch structures: there are more paths through Replace than the other three programs put together and the majority of TCAS' program code is not executed unless two numerical equalities, two inequalities and two Boolean values are true. Subdomain optimisation works best on programs for which it is possible to define a continuous effective region of test input: e.g. finding subdomains that contain various types of triangle for TriTyp, or optimising relative priorities for Schedule.

Table 3.4: Comparison with Dynamic Symbolic Execution

Program	**Mutation score**		
	Subdomain optimisation *	DSE 1 [65] †	DSE 2 [128] ‡
TCAS	47%	64%	54%
TriTyp	95%	69%	59%
Schedule	85%	57%	57%
Replace	43%	56%	53%
Program	**Number of test cases**		
	Subdomain optimisation *	DSE 1 [65] ‡	DSE 2 [128] †
TCAS	100	422	*unknown*
TriTyp	100	90	*unknown*
Schedule	100	301	*unknown*
Replace	100	8927	*unknown*

* averaged over 100 trials, † result of a single trial, ‡ averaged over 10 trials

† mutants produced using MuJava ‡ mutants produced using another tool (Milu)

Papadakis and Malevris [128] used 8927 test cases to achieve a 56% mutation score with Replace, whereas optimised subdomains achieved a 43% score with 100 test cases. Subdomain optimisation evaluates test cases at each generation to evaluate new candidate subdomain values. Table 3.5 lists the number of generations it took for each program before convergence was achieved (the point at which the mutation score no longer increases). It took an average of

244 generations to achieve convergence with Replace, so subdomain optimisation evaluated almost three times as many test cases as DSE. The real benefits of the new technique only become available once the subdomains have been evolved.

Table 3.5: Number of Generations Before Convergence Using 100 Test Cases

Program	**Number of generations**	
	Maximum	*Average*
Power	36.0	5.68
TrashAndTakeOut	216	79.4
FourBalls	197	75.8
TCAS	297	183
Cal	297	203
TriTyp	300	207
Schedule	269	65.1
Replace	476	244

Figure 3.1 plots the mutation scores achieved by subdomain optimisation, along with the expected scores for random testing (see Section 3.5.1). The same number of test cases are used in random testing as are sampled from the evolved subdomains (as opposed to the entire optimisation process). There is a relationship between the size of a program and the mutation score achieved by subdomain optimisation. With 10 test cases sampled from optimised subdomains, mutation score is correlated to the number of mutants and lines of code with -0.690 and -0.667 Spearman's rank coefficients. There is also a correlation (though a slightly weaker one) for other sizes of test suite. It is possible to kill most of the mutants from the smallest program (Power) with 10 test cases and little optimisation (see Figure 3.1a). In contrast, the largest program (Replace) had a low mutation score, even after 600 generations and 1000 test cases (see Figure 3.1h).

On the other hand, a program's size does not always determine how many of its mutants can be killed. Cal has twice the number of mutants as Fourballs, but 89% were killed by 10 test cases, compared to 83% with FourBalls (see Figures 3.1e and 3.1c). TriTyp has one more line than TrashAndTakeOut, but only 64% of its mutants are killed (see Figures 3.1f and 3.1b). The mutation score achieved with TCAS was only slightly higher than that achieved with Replace, but it has one sixth the number of mutants and one quarter the lines of code.

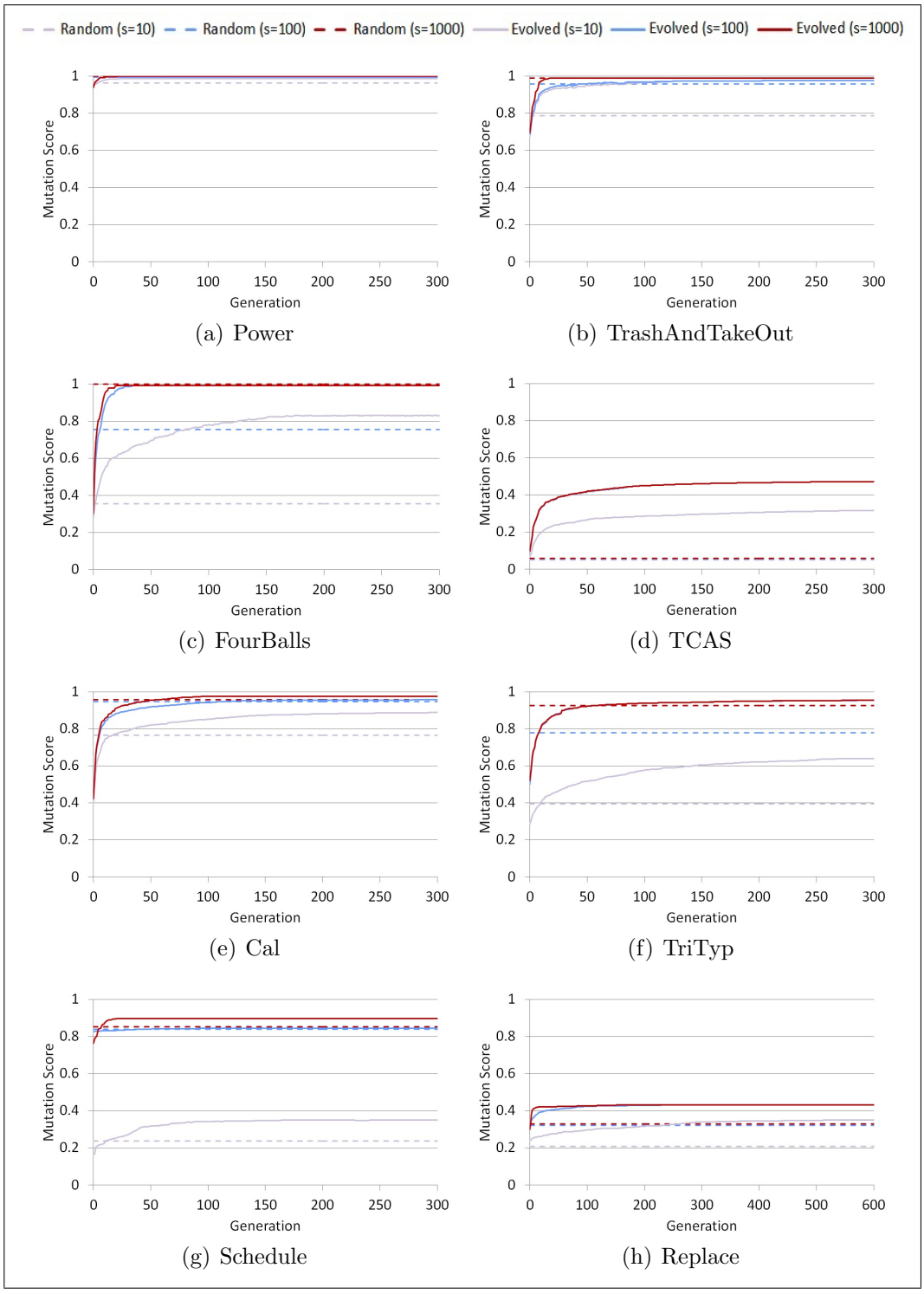


Figure 3.1: Mutation Scores for Random Test Suites and Evolved Subdomains (Averaged over 100 Trials)

None of the original trials with TCAS were able to produce a mutation score above 0.05, the score predicted for random testing (see Figure 3.1d). Inspection of the program code reveals TCAS uses large constants in equality conditions. For example, unless the value of $Cur\_Vertical\_Sep$ is greater than 600, most of the code will not execute. The mutation score was improved slightly by widening the initial subdomains, but it was more productive to scale the program constants. The program was transformed by dividing eight of its constants by 10, thus bringing them within the 0-100 range used for the initial subdomain limits. With the transformed program, the optimised programs achieved an average mutation score of 0.316 with 10 test cases, 0.470 with 100 test cases and 0.471 with 1000 test cases (see Figure 3.1d, NB: the s=100 line is covered by s=1000).

Subdomains discovered on the transformed program can be scaled up for use on the original program by multiplying the relevant values by 10. The subdomains identified by the technique were scaled to achieve an average mutation score of 0.401 for 1000 test cases, with one of the trials achieving 0.625. This is comparable to the 0.643 mutation score achieved by Papadakis et al. [128] with dynamic symbolic execution. The approach could easily be applied to other programs by manually identifying the relationship between the input parameters and the internal program constants to determine which parameters should be scaled. This could however be time consuming for a human tester who needs to test more complex programs. In Chapter 4, I introduce an automated program stretching technique to address this problem and achieve even better results.

Summary for RQ1: Test suites sampled from optimised subdomains were found to be at least as efficient as unoptimised random testing. In particular:

1. Optimised subdomains increased the mutation score compared to manually selected subdomains whenever the mutation score was not already 100%
2. There is a correlation between the size of a program and how difficult its mutants are to kill, with some exceptions (TCAS is small but challenging)
3. The mutation score for TCAS was increased from 0.05 to 0.401 by scaling its internal constants (and then de-scaling the optimised subdomain values)
4. Subdomain optimisation produces comparable results to dynamic symbolic execution and presents the tester with fewer test cases to evaluate

3.5.2 Results for RQ2

RQ2 was answered by optimising subdomains with three different shapes of input distribution (uniform, normal and exponential). I sample test suites of 10 test cases because this was previously shown to highlight the difference between test suite sizes clearly. Figure 3.2 shows a significant difference in average mutation score between the shapes of input distribution. Normal distributions achieved higher mutation scores with TCAS, Cal and Schedule; exponential distributions performed better with Power, TrashAndTakeOut, FourBalls, TriTyp and Replace.

Even though the mutation score differs from one shape of input distribution to the next, this difference is very small compared to the improvement made by optimising the subdomains themselves. The biggest improvement in mutation score (13.4%) was seen when changing the FourBalls sampling distribution from uniform to exponential. This is much less than the improvement already made for this program (134%) by optimisation from its initial subdomains. The difference is even smaller when evaluating subdomains with a greater number of sampled test cases. There is very little difference between distribution shapes when sampling 100 test cases from FourBalls and no significant difference was observed between the distribution shapes used to sample 1000 test cases. Overall, different shapes of subdomains have a minimal effect on the mutation score of sampled test suites.

It can also be difficult to determine in advance which shape of input will be the most effective. Of the eight programs evaluated, uniform sampling never achieved the highest mutation score. This suggests it is more effective to focus on a particular part of each subdomain. Normal distributions performed better on some programs and exponential distributions on others, but it is difficult to determine when each distribution will work best. There is no clear strategy (that can be determined from these results) for choosing the right distribution shape.

Summary for RQ2: Different input distribution shapes do have a small effect on the mutation score of sampled test suites. However, this effect is small compared to the improvement made by subdomain optimisation and no one shape can be considered the most effective for every program. It therefore seems unproductive to continue this line of research. Neither uniform, normal or exponential distributions match the ideal shape for each program. I achieve more success in Chapter 4 by using multiple sets of subdomains for greater precision.

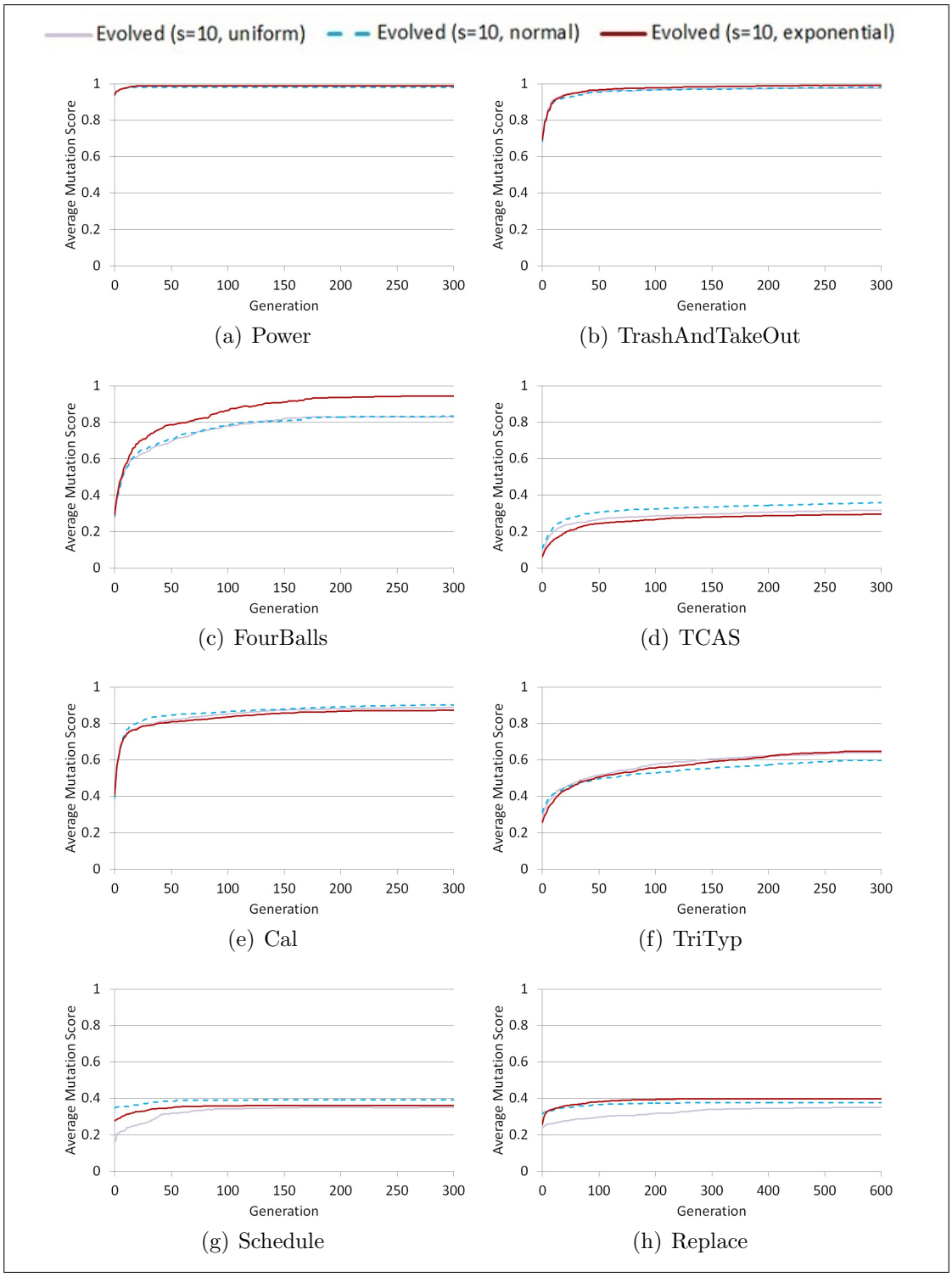


Figure 3.2: Mutation Scores for Subdomains with Different Distributions (Averaged over 100 Trials)

3.5.3 Results for RQ3

RQ3 was answered by examining the relationship between subdomain boundary (and chance) values and the mutation scores that they achieve. Throughout the process of subdomain optimisation, information is recorded on the mutation scores achieved by test suites sampled from various sets of subdomains. This information is used by the evolution strategy to discover and identify more efficient subdomains, but it can also be used to reveal useful information about the characteristics of the program under test. For example, it can be used to predict the existence of control branches in the program code or determine the necessary thresholds for input parameters to achieve a high mutation score. I present this information by providing several illustrating examples of ways in which subdomains characterise certain elements of the program code.

The Power program inputs two integers (x and y), then returns the value of x^y by applying $y - 1$ multiplications of x. If the value of y is less than or equal to zero, Power does not enter its multiplication loop, instead returning the value of x. As the majority of mutable statements occur in or around this loop, most of the mutants will not be exercised unless positive values for y are generated. For this reason, upper boundaries of y that are less than or equal to zero produce low mutation scores (see Figure 3.4). Setting the lower boundary of y to a positive value prevents negative numbers being generated and typically produces a mutation score around 95% (see Figure 3.3). Yet, in order to exercise all the mutants, it is necessary to include at least one negative and one zero value. 100% mutation score is only achieved if the lower boundary of y is negative.

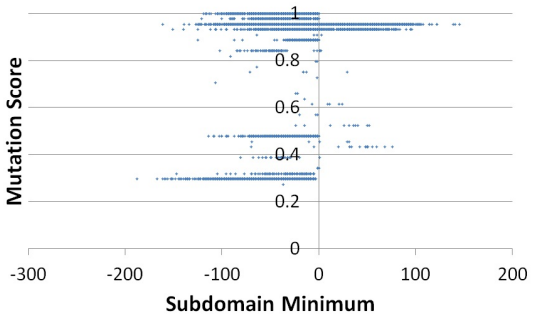


Figure 3.3: Lower Boundaries for 'y'

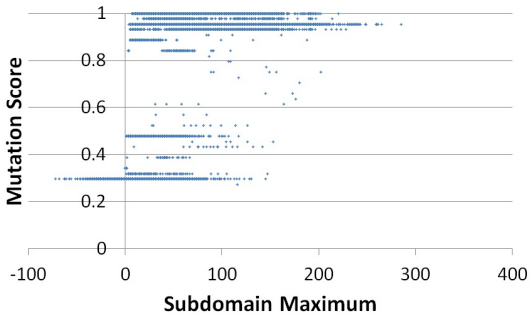


Figure 3.4: Lower Boundaries for 'y'

TrashAndTakeOut only has one input parameter, 'x', but two branch conditions $x > 0$ and $x > 5$. To achieve 100% mutation score, the lower boundary must be less than zero (see Figure 3.5) and the upper boundary greater than five (see Figure 3.6). As with Power, a positive lower boundary avoids low mutation scores, but in this case only achieves 66% mutation score.

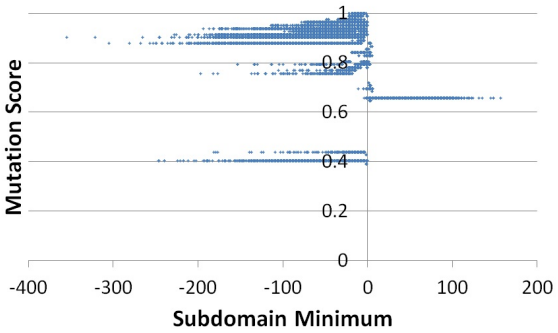


Figure 3.5: Lower Boundaries for 'x'

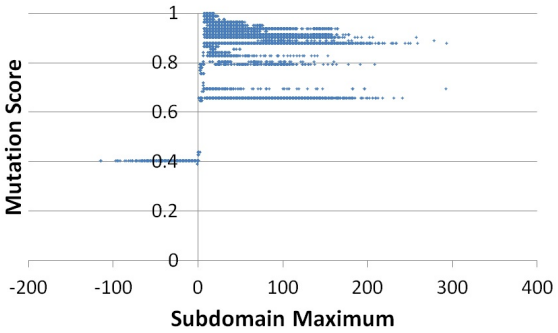


Figure 3.6: Upper Boundaries for 'x'

TCAS has a large number of input parameters, each of which have a different effect on the program. To achieve a high mutation score, the upper boundary of 'Cur_Vertical_Sep' must be greater than 60 (see Figure 3.7). This corresponds to a (global constant) threshold condition of 600 in the untransformed program code. It is also important to have a large chance for 'High_Confidence' to be true, as much of the code is not executed if it is false (see Figure 3.8).

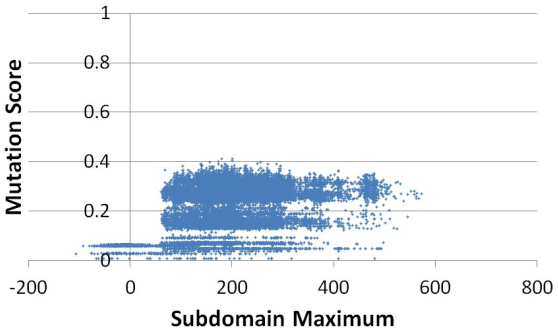


Figure 3.7: Upper Boundaries for 'Cur_Vertical_Sep'

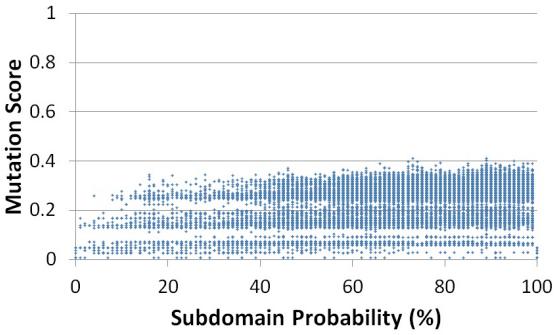


Figure 3.8: Chance values for 'High_Confidence'

Before starting this research, I assumed TriTyp would require small subdomains of integers close to zero to increase the likelihood of isoceles, equilateral and invalid triangles. In reality, there is little pressure towards the use of smaller subdomains (see Figure 3.9 and 3.10). It is only necessary for the upper boundary to be large enough to support each type of triangle. This appears to contradict the findings of Michael et al. [107] and Duran [44]. Yet, all of the subdomains evaluated in this research are relatively small, so it may still be valid that sampling from small subdomains is more efficient than the entire input domain.

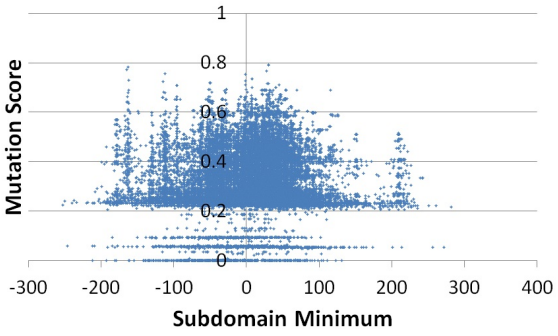


Figure 3.9: Lower Boundaries for 'side1'

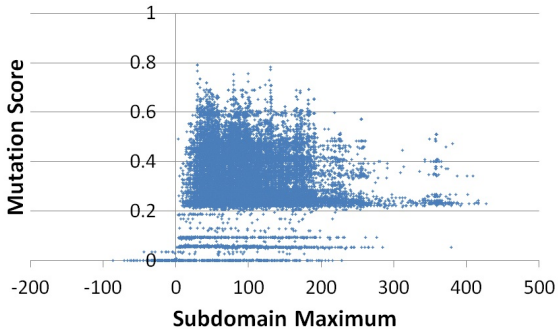


Figure 3.10: Upper Boundaries for 'side1'

Subdomain optimisation can be used to predict branch structure. The results of FourBalls show four distinct levels of mutation score (see Figure 3.11 and 3.12). They correspond to four branches in the program code, conditioned upon the value of 'cual' (1, 2, 3 or other). Figure 3.11 suggests the 'cual' subdomain must be small to achieve a high mutation score (values greater than three produce the same result). Compare this with the apparent correlation for 'a' that is actually just an artifact of the normal distribution (see Figure 3.12). It is important to compare any inferences against the expected results for an unguided search.

Many of the input parameters evaluated had little impact on the mutation score because they do not affect the control flow. Take for example the base values of Power compared with the exponent values. Input parameters can also be ineffective when their subdomains are poorly coded to parameters of the evolution strategy. My coding of the Replace program was unhelpful because it is irrelevant which alphabetical characters are used; only the special characters are significant. It is difficult to design an effective coding system, but the results of subdomain evaluation can often reveal new useful information for this task.

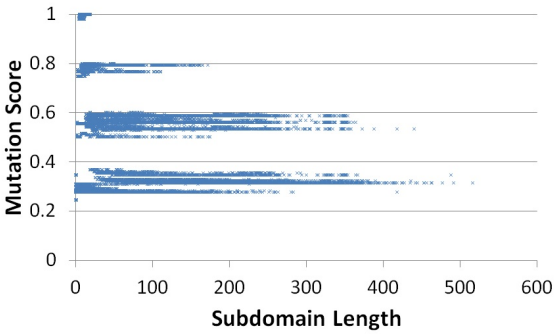


Figure 3.11: Size of Subdomains for 'cual'

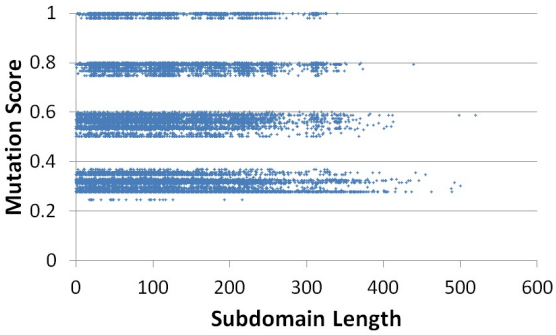


Figure 3.12: Size of Subdomains for 'a'

I have shown the subdomain optimisation technique presented in this chapter to be capable of revealing information about the programs under test, but it has at least one significant limitation. If values in the input domain necessary for killing mutants are spaced far apart, the highest mutation score will be achieved when the subdomain includes all these values. Yet, widening the subdomain has a negative effect because the likelihood of sampling these values is reduced. Subdomain optimisation is torn between widening the subdomains to make it possible to kill more mutants, or focussing on an efficient area of the input domain.

An illustration of this limitation can be found in the results for Schedule. Figure 3.14 shows that the highest mutation scores are achieved when the subdomain for 'prio_1' is made larger. There are useful areas within the input domain for this parameter at around 500 or -500 (see Figure 3.13). Yet, due to the likelihood of sampling these values from such as large subdomain, many of the evaluations produce low mutation scores. In Chapter 4, more mutants are killed with a smaller test suite by evolving multiple sets of subdomains, one for each group of mutants.

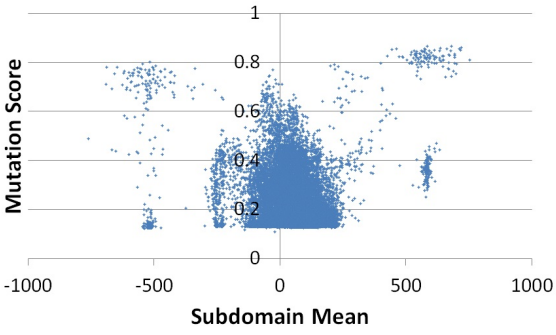


Figure 3.13: Centre Points for 'prio_1'

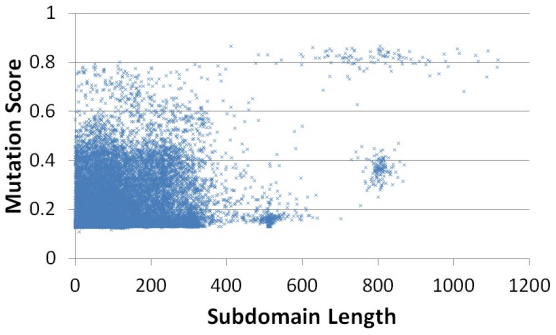


Figure 3.14: Size of 'prio_1' Subdomains

Finally, in addition to creating scatter plots of the evaluated subdomain values and their corresponding mutation scores, it is also possible to produce 3D representations of the optimisation process. Figures 3.15 through to 3.22 plot for each program the percentage of trials that achieved a particular mutation score at each generation in the evolution strategy. I ran 100 trials of my experiments and partitioned the mutation scores into intervals of 0.1 for the purpose of plotting my results. Rather than suggesting specific values to be used in testing (as with the scatter plots), 3D graphs provide insight into the optimisation process.

It is immediately apparent from the graphs that there is a distinction between programs for which all the mutants are killed straight away (e.g. Power) and programs for which many of the mutants are difficult to kill (e.g. TCAS). Since high mutation scores are achieved quickly for Power and many of the TCAS mutants are not killed, there are large flat areas in their graphs where there is little or no selection pressure. By contrast, programs reveal the most useful information for testing if their mutants are difficult but still possible to kill, as their subdomains are highly specialised by the end of the optimisation process.

For some programs (e.g. TriTyp) optimisation progresses smoothly from start to finish, whereas for others (e.g. FourBalls) there are peaks or 'ripples' in mutation score where optimisation has become stuck in a local optimum. This distinction can also be seen in the scatter plots to a lesser extent. Ripples correspond to branch conditions that are difficult to meet. One way to address this problem is to restart a trial once it has become stuck. Another (less disruptive) way is to modify the fitness landscape so the trial is no longer stuck. I do this in Chapter 4 by transforming the branch conditions to make them easier to meet.

Summary for RQ3: The relationship between subdomain values and the mutation scores they achieve can be characterised through the use of graphs. In particular, the following provide information about program branch structure:

1. 3D plots of the optimisation process (with ripples in mutation score) can be used to infer the number of branches in a program

2. Scatter plots of subdomain lower and upper boundaries reveal thresholds that must be met to satisfy particular branch conditions

3. Scatter plots of subdomain sizes indicate when a narrow range of values is needed to exercise branch conditions efficiently

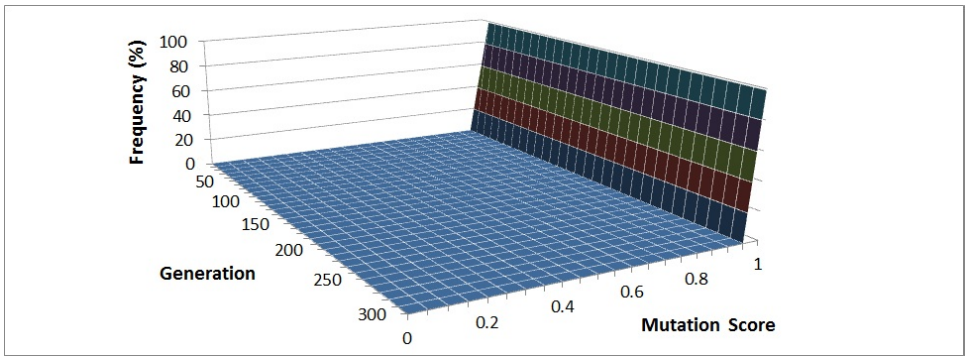


Figure 3.15: Power Optimisation Process

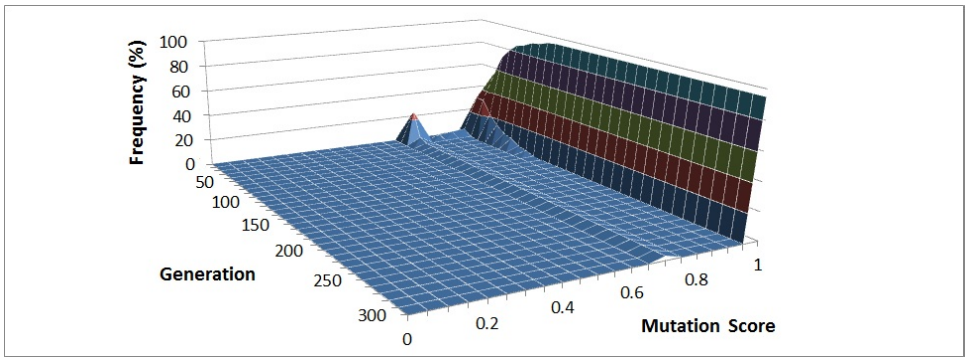


Figure 3.16: TrashAndTakeOut Optimisation Process

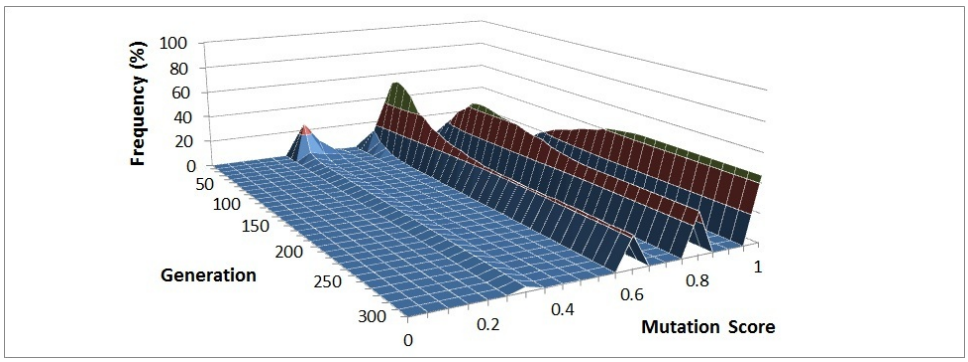


Figure 3.17: FourBalls Optimisation Process

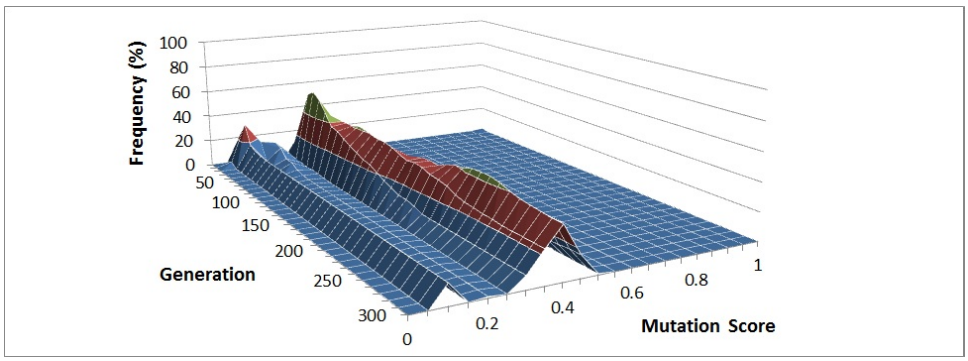


Figure 3.18: TCAS Optimisation Process

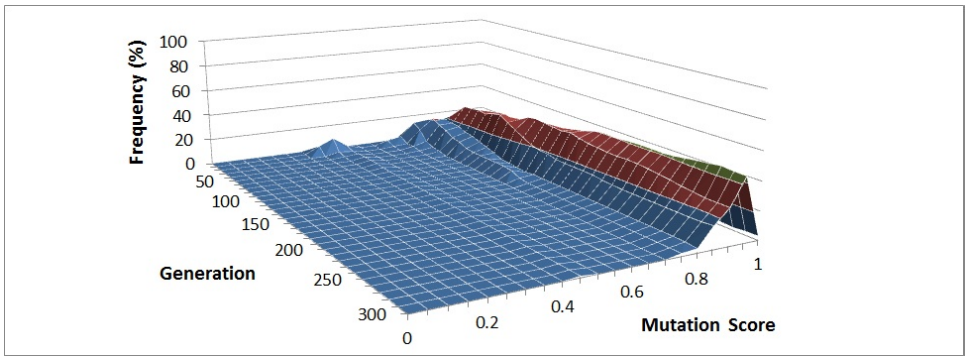


Figure 3.19: Cal Optimisation Process

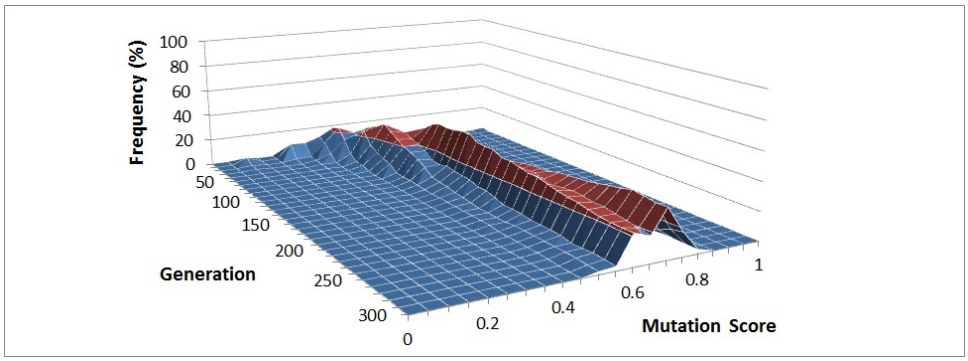


Figure 3.20: TriTyp Optimisation Process

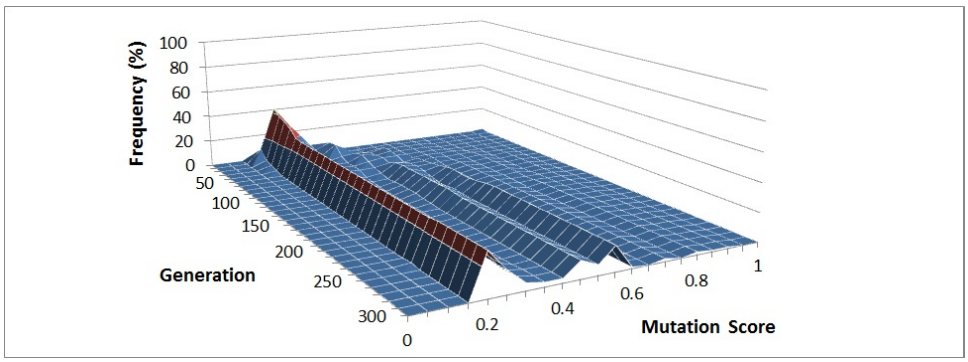


Figure 3.21: Schedule Optimisation Process

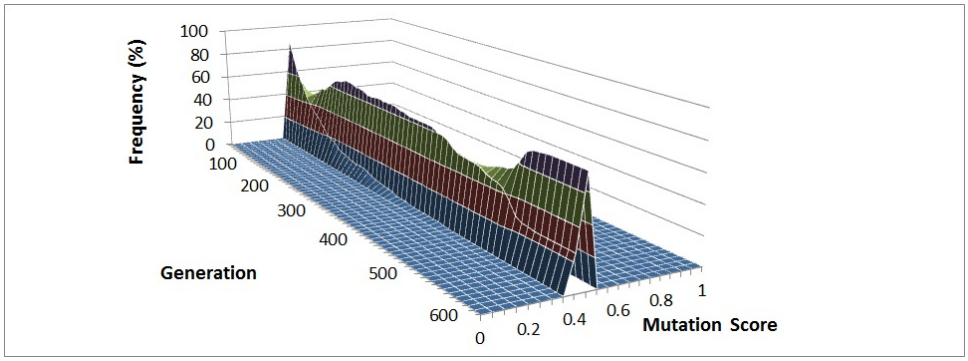


Figure 3.22: Replace Optimisation Process

3.6 Summary

The optimisation technique identified subdomains from which test cases can be selected with higher mutation score than random testing (numerical: [0,100], Boolean: 50%, character array: whole alphabet). This was achieved for eight benchmark programs, three sizes of test suite and three shapes of input distribution. It only failed to surpass the expected mutation score when it was 100%.

Optimisation increased the mutation score of each test suite, but greater improvements were made using 10 test cases compared to 100 or 1000, largely because the initial mutation score was much lower. The three shapes of input distribution used to sample test cases had little effect on the mutation score. The distribution shapes used in our experiments are primitive. This allows different distribution shapes to be specified without any additional parameters, but Chapter 4 shows that more complex input distributions achieve better results.

Scaling the parameters of TCAS allows more effective identification of subdomains. Some basic understanding of the program code was necessary to determine which parameters to scale. This threatens the validity of subdomain optimisation as a black-box technique, but it was only necessary to have knowledge of the global constants in TCAS. No scaling was needed for the other seven programs.

The strengths of the subdomain optimisation technique are:

1. **It allows black-box testing for an unknown program**
2. **It selects subdomains as a starting point for regression testing**
3. **It provides some insight into the thresholds and emphases required for choosing effective test input values (e.g. for TCAS)**

The weaknesses of the subdomain optimisation technique are:

1. **It achieves a lower mutation score than DSE for some programs.**
2. **It is inefficient when the values that kill each mutant are far apart.**
3. **It sometimes necessary to inspect the program code before optimising subdomains so as to identify appropriate scaling factors.**

These weaknesses are addressed in the next chapter by evolving multiple sets of efficient subdomains and stretching the program code automatically.

Chapter 4

Efficient Sets of Subdomains for Mutation Adequacy

4.1 Introduction

In the previous chapter, I evolved candidate solutions for the program under test with a single subdomain for each input parameter. The evolved subdomains achieved a higher mutation score than can be expected by generating test cases at random, but there is still some room for improvement. Many of the experimental trials became stuck in a local optimum rather than reach their full potential and one program (TCAS) required manual scaling of its global constants before subdomain optimisation could be effective. In this chapter, I evolve multiple sets of subdomains, each of which are targeted at killing a specific group of mutants as efficiently as possible. I also apply subset selection to identify small sets of subdomains that achieve mutation scores almost as high as the complete set.

As a motivating example, consider a program with two mutants ($M1$ and $M2$) and one input parameter (x); $M1$ can only be killed if $x = 1$ and $M2$ can only be killed if $x = 1000$. The smallest single subdomain for x capable of killing both $M1$ and $M2$ is the interval $[1, 1000]$. Widening the subdomain to this interval reduces the efficiency of the sampled test suite so the probability of killing each mutant is $1/1000$. Rather than evolving a single subdomain for both mutants, it is more efficient to evolve separate subdomains, one for each mutant. Subdomains can be evolved individually for the highest possible probability of killing each mutant.

However, evolving subdomains for individual mutants carries the risk of overfitting. This can increase the human effort involved in evaluating test cases and limits the potential for regression testing. The key to selecting efficient sets of subdomains is to find a balance between targeting mutants individually and using as few subdomains as possible. I evolve sets of subdomains to complement each other by targeting different groups of (similarly related) mutants. Initially I train the subdomains against the complete set of mutants, then later in the optimisation process against mutants for which no effective subdomain has yet been found. The first subdomains to be identified are therefore reasonably good at killing a large number of mutants, whereas subdomains evolved later are more efficient at killing specific groups of mutants. By evolving some subdomains to target a large number of easy to kill mutants and others to target a small number of difficult to kill mutants, mutants can be killed efficiently with as few subdomains as possible.

Another way to improve efficiency and reduce the number of subdomains is to select subsets of subdomains once they have been produced. Previously (in Chapter 3), I found that reducing the number of test cases sampled from a single subdomain increased test suite efficiency without greatly reducing performance. In this chapter, I explore whether reducing the number of subdomains has a similar effect. I reduce the number of subdomains by means of a new sequential technique for subdomain selection. Subdomains are added and removed one at a time until the highest possible mutation score is achieved for each set size. Subdomain selection is applied incrementally, from a single subdomain up to the complete set. In this way, it is possible to identify the smallest set of subdomains that have a similar fault finding capability to the complete set.

Optimising efficient multiple sets of subdomains addresses the following problems of conventional random testing:

1. **It is inefficient for faults that require specific boundary conditions** – I evolve multiple sets of subdomains to target boundary conditions more efficiently than sampling over a single large subdomain.

2. **Without an automated oracle, random testing is labour intensive** – My technique requires fewer test cases than is typical for random testing, thus reducing the human effort required to create test oracles.

4.2 Optimising Multiple Sets of Subdomains

Multiple sets of subdomains are optimised using an evolution strategy. The process (see Figure 4.1) is similar to that presented in Chapter 3, except it involves a more sophisticated evolution strategy (CMA-ES) and a new fitness function favouring subdomains that kill different groups of mutants. Subdomains are evaluated by sampling test cases within their bounds and counting the number of times each mutant is killed. Once a group of mutants is covered by a set of subdomains, the search continues with the remaining mutants.

A mutant is considered to be covered if it is killed at least once in 95 out of 100 test suites of 5 test cases sampled from the subdomains. This ensures that each set of subdomains consistently kills a different group of mutants. Once subdomains are found to cover a particular group of mutants, the search continues with the aim of identifying and targeting a new group from among the remaining mutants. If, however, no new mutants have been covered after 50 generations, the program is transformed through a process known as 'program stretching' to make individual mutants easier to kill one at a time.

Rather than increasing the mutation score continually for a single set of subdomains (as in the previous chapter), the technique presented in this chapter is discontinuous. It restarts the optimisation process whenever a group of mutants has been sufficiently covered by the current set of subdomains. The search is terminated if, after the stretching process is completed for all the remaining mutants, no further mutants have been covered.

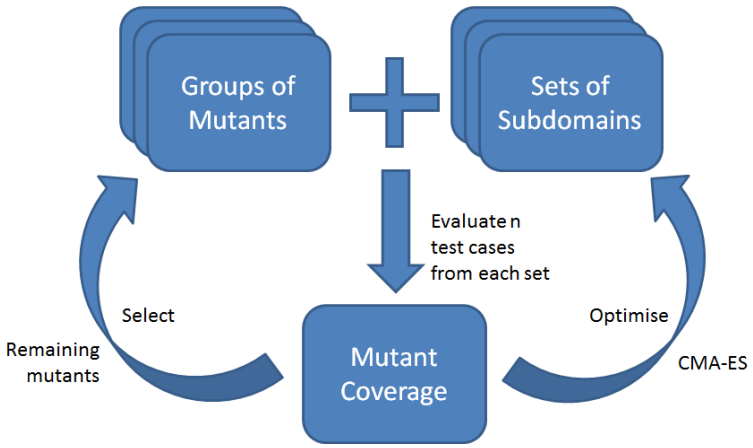


Figure 4.1: Optimising Multiple Sets of Subdomains

4.2.1 Covariance Matrix Adaptation Evolution Strategy

Covariance Matrix Adaptation evolution strategies (CMA-ES) [62] can solve difficult optimisation problems without the need for manual parameter tuning. CMA-ES works by adapting the search distribution at the same time as the candidate solutions. This means that the algorithm can adjust itself automatically to the fitness landscape. CMA-ES has been shown to be particularly effective at non-linear optimisation. In a recent black-box comparison study with 25 benchmark functions, CMA-ES outperformed eleven other algorithms in terms of the number of function evaluations before the global optimum value is reached [63].

CMA-ES defines the search neighbourhood using a multivariate normal distribution, represented in Figure 4.2 as a constantly changing oval. The distribution mean is set, as in the traditional form of evolution strategy, to the currently favoured solution. Yet, in contrast to the traditional form, which uses the same variance for each parameter, CMA-ES defines the shape of the distribution using a covariance matrix and scaling factor [62]. Multiple dimensions of variance allow the search distribution to be adapted precisely to the underlying fitness landscape. CMA-ES adjusts the shape and size of the distribution according to pairwise dependencies identified in the covariance matrix [62]. The search distribution is adapted to balance the exploitation and exploration of values for each input parameter, so as to achieve fast, but not premature convergence.

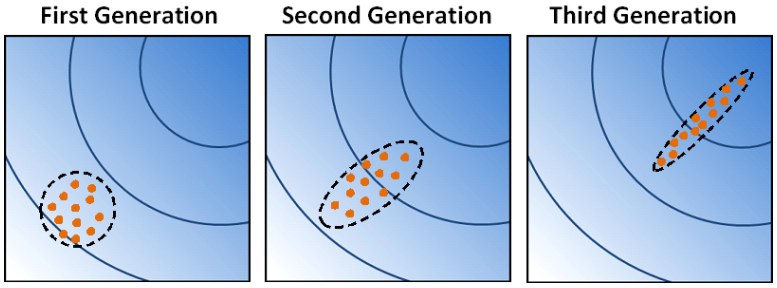


Figure 4.2: Covariance Matrix Adaptive Evolution Strategy (CMA-ES)

(CMA-ES is a population-based optimisation algorithm.
The solid blue lines represent points of equal fitness in the landscape,
the dashed lines represent the multivariate normal search distribution
and the orange dots represent sampled candidate solutions)

4.2.2 Fitness Function for Evolving Sets of Subdomains

As well as employing a different form of evolution strategy (CMA-ES), I also introduce a new fitness function with the aim of selecting subdomains which consistently kill the same specific group of mutants (see Equation 4.1). By targeting each set of subdomains at a different group of mutants, test data can be sampled that is able to kill each group as efficiently as possible.

$$\text{Minimise} \sum_{s \in S} \sum_{m \in M} \frac{(K_{s,m} - \bar{K}_m)^2}{(\bar{K}_m - \bar{K})^2} \tag{4.1}$$

(S is the set of test suites, M is the set of mutants, K is the number of kill events)

The fitness function in Equation 4.1 does not require the user to specify which particular group of mutants to target. Instead, it maximises variance in the number of times each mutant is killed and minimises variance in the number of times the same mutant is killed. This metric is similar to the lack-of-fit sum of squares calculation used in an F-test, for example to measure homogeneity in cluster analysis [18]. The fitness function consists of a sum of fractions. Each numerator represents the pure-error sum of squares (differences within the same mutant) and each denominator represents the lack-of-fit sum of squares (differences between mutants). By minimising the fitness function, the CMA-ES will tend to select subdomains that consistently kill the same group of mutants. The end result is that each set of subdomains is trained against a different group of mutants.

4.2.3 Subdomain Representation

Subdomains are expressed in the same forms as they are in the previous chapter, but with a different character array representation (see Table 4.1). Rather than optimise upper and lower boundaries for characters as if they are numerical values, the new approach assigns a separate chance of selection to each special character (wildcard, closure etc.) and a single chance of selection to alphabetical characters (a-z). Character arrays are fixed in length (by default to five characters) and the value of each character is sampled randomly. Characters are sampled from a weighted distribution, as determined by the evolved probabilities.

Table 4.1: Transformations from $x \in [0, 1]$ to values within a subdomain

Data type	**Previous representation**	**New representation**
Numerical (n)	$n = x * (n_u - n_l) + n_l$	$n = x * (n_u - n_l) + n_l$
Boolean (b)	**if**$(x < b_p)$ $b = true$, **else** $b = false$	**if**$(x < b_p)$ $b = true$, **else** $b = false$
Character (c)	$c = x * (c_u - c_l) + c_l$	**for each** $i \in character\_set$ **if**$(x < c_{p(i)})$ $c = c_i$

(n_u and n_l are the upper and lower boundaries for a numerical subdomain, b_p is the probability of a value within a Boolean subdomain being true, c_u and c_l are the upper and lower boundaries for a character subdomain, and $c_{p(i)}$ are probability thresholds for specific characters within a subdomain)

4.2.4 The Core Optimisation Algorithm

Algorithm 5 outlines the main process used to identify efficient subdomains for a program that has numerical input parameters. It starts by selecting n initial random means for the upper and lower range of each parameter, then adapts these values until they are efficient (as determined by their fitness evaluation) at killing a specific group of mutants. A similar process is applied with Boolean probabilities and character array distributions, except that subdomain values are selected using percentage chance values rather than upper and lower boundaries.

The first set of subdomains is evolved by applying the optimisation algorithm to all the mutants from the program under test. The algorithm iterates, by using its fitness function to select more efficient subdomains, until a specific group of mutants is killed by more than 95 sampled test suites out of 100. At each iteration, the default $4 + 3 \log n$ [62] new candidate values (Λ) are perturbed from the current means and their fitness is evaluated. The fittest 50% of these values (Υ) are selected, weighted according to fitness and averaged to produce means for the next iteration. At the same time, the step size (σ) and covariance matrix (C) is updated to optimise the search distribution. The covered mutants are then put to one side and the boundary values of each subdomain recorded. The process is repeated to evolve new subdomains until at point is reached at which no more subdomains can be identified to cover the remaining mutants.

Algorithm 5 Synthesising an optimal solution ($[\alpha_l, \alpha_u]$, $[\beta_l, \beta_u]$, ..., $[\Omega_l, \Omega_u]$)

1: Select initial random values (uniformly from the range $0 \ldots 100$) for the means ($\mu_1 \ldots \mu_n$) and step size (σ), where n is twice the number of subdomains
2: Initialise the covariance matrix (C) to give equal emphasis to each direction
3: Set the sample size (Λ) to $4 + 3 \log n$ and the selection size (Υ) to $\Lambda/2$
4: **loop**
5: **for** $\lambda \in \Lambda$ **do**
6: Sample candidate values from a multivariate normal distribution:
 $x_1^\lambda = \mu_1 + \sigma\epsilon_1$, $x_2^\lambda = \mu_2 + \sigma\epsilon_2$, $x_3^\lambda = \mu_3 + \sigma\epsilon_3$, $x_4^\lambda = \mu_4 + \sigma\epsilon_4$,
 ..., $x_{n-1}^\lambda = \mu_{n-1} + \sigma\epsilon_{n-1}$, $x_n^\lambda = \mu_n + \sigma\epsilon_n$ where $\epsilon_1 \ldots \epsilon_n \in \mathcal{N}(0, C)$
7: Generate 100 suites of 5 test cases from $[x_1^\lambda, x_2^\lambda]$,$[x_3^\lambda, x_4^\lambda]$,...,$[x_{n-1}^\lambda, x_n^\lambda]$
8: **for** $m \in M$ **do**
9: **for** $s \in S$ **do**
10: Count the number of times ($K_{s,m}$) mutant m killed by test suite s
11: **end for**
12: $\bar{K}_m = \sum_{s \in S} K_{s,m}/100$
13: **if** $\sum_{s \in S} Kills(m, s) \geq 95$ **then**
14: **return** $[x_1^\lambda, x_2^\lambda]$,$[x_3^\lambda, x_4^\lambda]$,...,$[x_{n-1}^\lambda, x_n^\lambda]$
15: **end if**
16: **end for**
17: $\bar{K} = \sum_{m \in M} K_m/|M|$
18: $Fitness_\lambda = \sum_{s \in S} \sum_{m \in M} \frac{(K_{s,m} - \bar{K}_m)^2}{(K_m - \bar{K})^2}$
19: **end for**
20: Sort the Λ samples by their fitness and select the Υ best samples
21: Calculate the selected sample weights (w_υ, where $\upsilon \in \Upsilon$)
22: Update the search distribution mean:
 $\mu_1 = \sum_{\upsilon \in \Upsilon} w_\upsilon x_1^\upsilon$, $\mu_2 = \sum_{\upsilon \in \Upsilon} w_\upsilon x_2^\upsilon$, ..., $\mu_n = \sum_{\upsilon \in \Upsilon} w_\upsilon x_n^\upsilon$
23: Update the covariance matrix (C) and the step size (σ)
24: **end loop**

(M is the set of non-equivalent mutants and S is the set of test suites.
$Kills(m, s) = true$ if $K_{s,m} > 0$ and $false$ otherwise)

4.2.5 Program Stretching

My technique for program stretching was inspired by previous research into the dynamic transformation of programs to improve branch coverage [53]. In order to adapt this concept to mutation analysis, I have designed new program transformations that allow difficult to kill mutants to be reached, infect a difference in the program data state and propagate this difference to the output. Stretching a program involves transformation of its code and a new fitness function. Instead of targeting a group of mutants (as was described in the previous subsection), I maximise the number of times an individual mutant is killed.

Program stretching makes specific mutants easier to kill by introducing 'delta' values into the program code. Once it is applied, program stretching can be gradually reversed by incrementally adjusting the delta values from 100 down to zero. By restoring the transformed program back to the original mutant step by step, program stretching aims to drag the subdomain values along with it so that subdomains are evolved that can efficiently kill the original mutants.

The following three 'stretch' modes are used in the research for this chapter:

Path stretching

forces branch conditions leading up to a mutant to be true or false, depending on whether the branch was taken the last time the mutant was killed. Path stretching is the first transformation to be applied. It is designed to train subdomains so that program execution reaches the point of mutation.

Mutation stretching

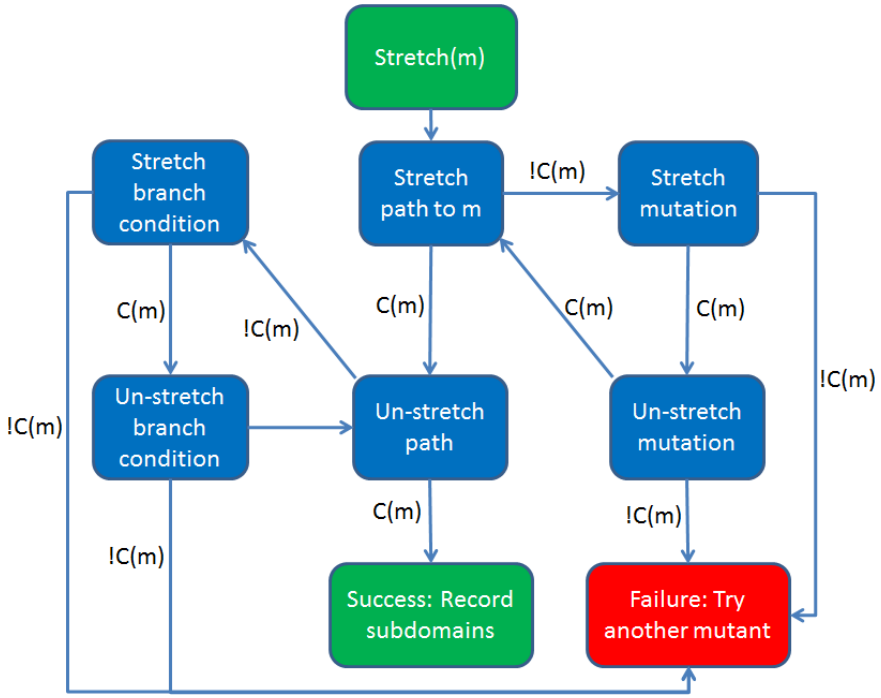
alters the mutation by an offset of 100, for example $x >= y \rightarrow x > y$ becomes $x > y + 100$ with the aim of increasing its impact on the program. Mutation stretching applied if subdomains still cannot be found to cover the mutant, even after the path to the mutation point has been forced.

Branch condition stretching

adds an offset of 100 to a difficult branch condition in order to make it easier to meet, for example $x == y$ becomes $(x <= y + 100)\&\&(y <= x + 100)$. Branch condition stretching is applied if restoring a branch condition following path stretching prevents the mutant from being covered.

The three program transformations were designed to make it easier to reach, infect and propagate the targeted mutants. Path stretching addresses reachability by forcing execution down a particular subpath, then evolving subdomains to reach the remaining branches in the path. Mutation stretching addresses the infection condition by attempting to increase the effect of the mutant (a linear delta may not have linear effect, so this is an approximation). Branch condition stretching addresses propagation by making difficult to meet branch conditions easier along paths on which the mutant has been killed.

Figure 4.3 summarises the process used to find effective subdomains for difficult to kill mutants by stretching and un-stretching the program code. Program stretching dynamically alters the fitness landscape so as to make the necessary subdomain values more readily available to the search process. It is performed so that the mutant that has been killed the most number of times is targeted first, then the next most frequently killed and so on. Once stretching is completed, the main fitness function is reapplied to take advantage of the stretching process on other mutants that are killed by similar input values.



(C(m) is true if the evolved subdomains cover mutant m)

Figure 4.3: Flowchart of the Program Stretching Process

4.2.6 Experiments

I set up experiments to answer the following two research questions in regard to optimising multiple sets of subdomains for mutation adequacy:

RQ4: Are test suites sampled from multiple sets of subdomains more efficient at killing mutants than single sets of subdomains?
A test suite can be considered to be more efficient if it achieves a higher mutation score with the same number of test cases. I use mutation analysis to determine whether multiple sets of subdomains are more efficient than single sets. This is achieved by sampling 5 test cases from each set of subdomains evolved as part of a multiple set, then sampling the same total number of test cases from single sets of subdomains and counting the average number of mutants killed by each approach. If multiple sets of subdomains kill more mutants than single sets with the same number of sampled test cases, multiple sets can be considered to be more efficient.

RQ5: Does the new approach for optimising multiple sets of subdomains take longer than the previous approach for optimising single sets?
Although test suites sampled from multiple sets of subdomains are expected to kill mutants more efficiently than those sampled from single sets, it may take longer for them to be evolved. Sets of subdomains are evolved one after the other in the multiple set approach, so that optimising a large number of sets can be computationally expensive. Yet, it is only necessary to cover one group of mutants at a time, so the stopping criterion for each set of subdomains is less demanding. It seems likely that a trade-off will need to be made between the amount of time required to evolve subdomains and the efficiency of test suites sampled from them once they have been evolved. The efficiency of the evolved subdomains is more important (since computation time is typically cheaper than human effort), but I will still take optimisation time into account. If multiple sets of subdomains take considerably longer to evolve than single sets and the mutation score they achieve is only slightly higher, I will consider the added value of using multiple sets to be questionable. If on the other hand, multiple sets of subdomains take slightly longer to evolve but achieve a significantly higher mutation score, I will consider the multiple sets approach to be worthwhile.

4.2.6.1 Methodology for RQ4

RQ4 is designed to compare the relative efficiencies of test suites sampled from multiple and single sets of evolved subdomains. Test suites sampled from single sets of optimised subdomains have previously been shown to achieve a higher mutation score on average than unoptimised random testing (see Chapter 3). The new technique for evolving multiple sets of subdomains should perform even better than the previous technique for evolving single sets of subdomains because it targets and optimises subdomains for individual groups of mutants. This avoids the problem that single sets of subdomains are made larger by the need to cover important values in different regions of the input domain, thus reducing their efficiency. Multiple sets of subdomains can target each key region of the input domain with a different set of subdomains for maximum efficiency.

I address this research question by applying the new technique to optimise multiple sets of subdomains on programs for which it was previously shown to be a non-trivial task to achieve a high mutation score with single sets of subdomains (see Table 4.2). Subdomains are optimised from their initial random starting values using a CMA-ES. The optimised subdomains are evaluated by sampling 5 test cases from each set, then counting the number of mutants that are killed by the overall test suite. I compare these results with my previous technique for optimising a single set of subdomains (with the same number of test cases). For completeness, the results are also compared with the mutation score expected to be achieved by this many random test cases sampled from [0,100] numerical intervals, 50% Boolean chance and the entire set of character values.

It will be interesting to observe the extent to which multiple sets of subdomains increase the mutation score above that achieved by single sets. Single sets of subdomains significantly improved the mutation score of many programs compared to unoptimised random testing. Even though previous results (see Chapter 3) suggest that multiple may be useful, it seems unrealistic to expect the same degree of improvement. It is more likely that multiple sets of subdomains will be more effective for some programs than others, depending on the values that are needed to kill their mutants. Nevertheless, any increase in mutation score is worthwhile, as long as it does not come with too much computational expense.

4.2.6.2 Methodology for RQ5

RQ5 is designed to investigate the additional computational expense involved with optimising multiple sets of subdomains compared to single sets. Answering this research question will help determine whether the increases in mutation score achieved by multiple sets of subdomains are worth the added expense. Multiple sets of subdomains are ultimately expected to achieve a higher mutation score than single sets, but (for a given mutation score that can be achieved by a single set) the process of optimising multiple sets of subdomains might be less efficient in terms of computation time. I expect the single set technique will be more efficient at achieving relatively low mutation scores, whereas the multiple set technique will be more efficient at achieving high mutation scores. There is likely to be a trade-off between the computational efficiency of the optimisation process and the human effort involved with evaluating the resulting subdomains.

I answer this research question using the same experiments described for RQ4, except that I record slightly different information from the results. In addition to comparing the mutation scores ultimately achieved by subdomains optimised using the multiple and single set approaches, I also determine the rate at which the mutation score increases as optimisation progresses. This does not require any further mutation score evaluations for single sets of subdomains. As optimisation is already performed on the basis of mutation score, it is simply a case of recording the new mutation score whenever it is improved by the optimisation process. For multiple sets of subdomains, however, optimisation is performed on the basis on mutant coverage rather than mutation score. It is therefore necessary to evaluate the mutation score whenever a new subdomain is added. I do this by sampling 5 test cases from each set of subdomains. The information gathered about the progress of mutation score over optimisation will help me to determine whether multiple sets or single sets take more time to achieve particular mutation scores.

4.2.6.3 Test Subject Programs

The new subdomain optimisation technique was applied to six programs (see Table 4.2), four of which were selected because they were the most challenging programs from the previous chapter. In my experiments optimising a single set of subdomains for each program, it was particularly difficult to produce efficient subdomains for TriTyp, Schedule, TCAS and Replace. This means that any differences between multiple and single set subdomains are likely to be more pronounced for these programs. The remaining two programs (SingularValueDecomposition and SchurTransformation) were chosen because they have a more complex (matrix) input data structure and have a large potential for mutation. It is hoped that these mutants will help to illustrate whether multiple sets of subdomains are more effective than single sets.

Table 4.2: Test Programs Used in the Experiments

Program	**Mutants**	**LOC***	**Function**
TriTyp	310	61	Triangle classification
Schedule	373	200	Task prioritisation
TCAS	267	120	Air traffic control
Replace	1632	500	Substring replacement
SingularValueDecomposition	2769	298	Matrix decomposition
SchurTransformation	2125	497	Matrix transformation

*LOC: Lines of Code

The subdomain representation for TriTyp, Schedule and TCAS is described in the Chapter 3. In contrast with this chapter, I evolve a probability of inclusion for each special character (wildcard) used by Replace. I also evolve numerical subdomains for each diagonal of a four-by-four matrix for SingularValueDecomposition and each value of a three-by-three matrix for SchurTransformation. These programs represent a variety of data structures and processing operations. Their mutants should also be more difficult to kill than those evaluated previously.

4.2.7 Results

Below are the experimental results, addressing each research question in turn.

4.2.7.1 Results for RQ4

RQ4 is answered by comparing the new approach for evolving multiple sets of subdomains with my previous approach (see Chapter 3) which evolves a single set of subdomains without program stretching. The results are presented graphically in Figure 4.4 and numerically in Table 4.3. Subdomains use the initial interval [0,100] (50% chance of being true for Booleans). In the interest of fairness, both techniques were evaluated with the same number of test cases (5 for each set of subdomains) and the results are averaged over 100 trials.

Multiple sets of subdomains achieved 33% higher mutation score on average than single sets and 230% higher than random testing (see Table 5.4). Although the difference in mutation score between the new technique and the previous one is small compared to the difference between either technique and random testing, there is still a definite improvement in mutation score. For all but one program (SchurTransformation), multiple sets achieved a higher mutation score than single sets. In this case, single sets and random testing already achieved a high mutation score, so there is little improvement that could be made. A Student's t-test at a 95% confidence interval reveals there is no significant difference in mutation score ($P = 0.244$) between single and multiple sets of subdomains for this program.

Table 4.3: Summary of Results (Averaged over 100 Trials)

Program	**Mutation Score**		**Time (mins)**		**Test Cases**
	Single	Multiple	Single	Multiple	
TCAS	0.457	0.780	364	50.6	205
TriTyp	0.951	0.998	78	8	135
Schedule	0.850	0.930	1053	1310	40
Replace	0.520	0.566	746	1410	455
SVD*	0.397	0.632	524	546	125
Schur†	0.986	0.920	958	885	45

∗ SingularValueDecomposition, † SchurTransformation

The new technique made a greater difference to the mutation score for some programs than others. I found it to be particularly effective at meeting difficult branch conditions. For example the TCAS program previously required manual scaling of its parameters (see Chapter 3). Multiple sets of subdomains and automated program stretching render this unnecessary, allowing a higher mutation score to be achieved. The new technique achieved a 71% increase in the mutation score for TCAS compared with the previous technique. Multiple sets of subdomains are particularly effective for difficult branch conditions, because they can assign a set of subdomains for the purposes of meeting each condition.

Test suites sampled from multiple sets of subdomains also performed substantially better than single sets in comparison with dynamic symbolic execution (see Table 4.4). Single sets of subdomains previously achieved a higher mutation score than dynamic symbolic execution for TriTyp and Schedule, but a lower mutation score for TCAS and Replace (see Chapter 3). The multiple set technique achieved a higher mutation score than dynamic symbolic execution for all four programs.

Table 4.4: Comparison with Dynamic Symbolic Execution

Program	**Mutation score**			
	Multiple sets *	Single sets *	DSE 1 [65] †	DSE 2 [128] ‡
TCAS	78%	46%	64%	54%
TriTyp	100%	95%	69%	59%
Schedule	93%	85%	57%	57%
Replace	57%	52%	56%	53%
Program	**Number of test cases**			
	Multiple sets *	Single sets *	DSE 1 [65] †	DSE 2 [128] ‡
TCAS	205	205	422	*unknown*
TriTyp	135	135	90	*unknown*
Schedule	40	40	301	*unknown*
Replace	455	455	8927	*unknown*

* averaged over 100 trials, † result of a single trial, ‡ averaged over 10 trials

Summary for RQ4: Test suites sampled from multiple sets of subdomains achieved 33% higher mutation scores than single sets on average. In particular, they achieved 71% higher mutation score than TCAS and do not require manual parameter scaling. Multiple sets were also considerably more effective than DSE.

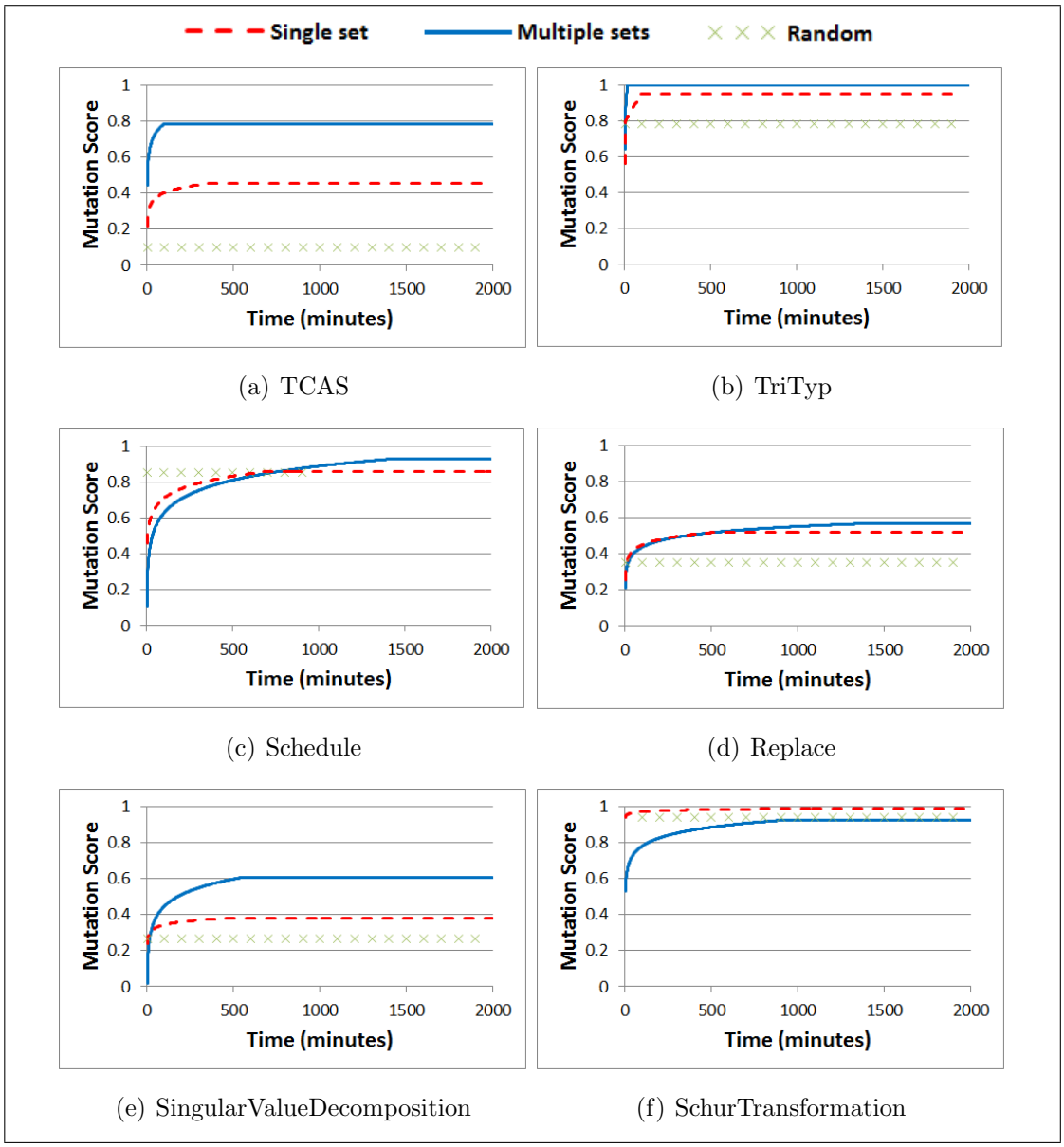
4.2.7.2 Results for RQ5

RQ5 is answered by comparing the amount of time it took to evolve single and multiple sets of subdomains and considering the progress of mutation score for the subdomains throughout their optimisation process. Figure 4.4 plots the mutation scores achieved by multiple and single sets of subdomains against the time taken to evolve them. The results are plotted to form an averaged continuous curve (across 100 trials) using least squares logarithmic curve fitting [143]. Logarithmic curve fitting makes it possible to compare the averaged results at each minute of computation time (between 0 and 2000 minutes). I correct the curve with a cut-off at the average point after which no further mutants were covered.

On average, it took 11.6% longer to evolve multiple sets of subdomains than single sets (see Table 5.4). Single sets of subdomains were evolved for the Replace program in just over half the time of multiple sets, but it took over seven times longer to evolve single sets for the TCAS program than multiple sets. It should be noted that these comparisons are based on the time taken until the last set of subdomains is identified (or there are no further improvements in mutation score). Typically, most mutants are killed quickly, but it then takes a long time to kill the few remaining mutants. It is therefore necessary to consider the mutation score achieved throughout the optimisation process, not just at the end.

Multiple sets of subdomains quickly achieve a higher mutation score than single sets on half of the programs (TriTyp, TCAS and SingularValueDecomposition). It still takes longer for SingularValueDecomposition to reach its maximum mutation score with multiple sets compared to single sets, but the final value is much higher. For reasons mentioned previously, multiple sets never achieve as high an average mutation score for SchurTransformation as single sets. On the two remaining programs (Schedule and Replace), multiple sets of subdomains perform similarly to single sets at first, but then eventually overtake it. The mutation score for Schedule is lower with multiple sets for the first 13 hours and it takes 9 hours for multiple sets to achieve a higher mutation score for Replace.

Summary for Q5: Multiple sets of subdomains take less time for some programs and more time for other programs to evolve than single sets. For the programs on which multiple sets of subdomains takes longer, the ultimately higher mutation scores outweigh the added computational cost.



(a) TCAS (b) TriTyp

(c) Schedule (d) Replace

(e) SingularValueDecomposition (f) SchurTransformation

Figure 4.4: Percentage of Mutants Covered by Evolved Subdomains (Averaged over 100 Trials)

4.3 Subdomain Set Selection

In the previous section, multiple sets of subdomains were optimised to achieve higher mutation scores than single sets, but this comes with a cost: sampling test cases from multiple subdomains produces a large test suite drawn from many different regions of the input domain. Appendix B.2 presents a k-means clustering of sets identified for the SingularValueDecomposition program. There are similarities between many of the sets, as those identified later in the optimisation process also kill mutants covered earlier. There are also some unique sets that have been evolved to kill specific, hard to kill mutants. The aim of this chapter is to select smaller sets of subdomains by removing those sets that cover mutants killed more efficiently by other sets, without losing any mutant killing capability.

Considerable research effort has been invested into finding effective ways of reducing test suites to minimise human and computational expense [158]. Proposed strategies include eliminating redundant test cases, selecting test cases relevant to recent changes and ordering test cases to reduce the time taken to find the first fault. Test suites are typically evaluated according to the coverage criteria met by each test case (e.g. the mutants they kill). Selecting subsets of subdomains is slightly different because test cases are sampled probabilistically. Existing test minimisation techniques necessitate an artificial threshold of coverage frequency. Although this is a valid option, it loses information about the frequency with which sets of subdomains kill particular mutants. I have devised an alternative selection approach that allows this information to be taken into account.

My approach borrows ideas from feature selection. Feature selection techniques can be roughly divided into two categories: optimal and suboptimal. Optimal techniques (e.g. branch-and-bound) are provably equivalent to exhaustive search, but computationally prohibitive. Suboptimal techniques do not guarantee the optimal selection of features, but are successful in the majority of cases and have much more modest computational requirements. My technique for subdomain set selection is based on a suboptimal feature selection. Suboptimal techniques typically employ greedy heuristics to quickly select features that provide the greatest improvement for a criteria evaluation. Amongst other applications, they have been used to diagnose Alzheimers disease from EEG data [3], detect emotion from speech [23] and determine steel quality from textural analysis [81].

4.3.1 Sequential Floating Forward Selection

Sequential Floating Forward Selection (SFFS) is a robust but suboptimal feature selection technique [134]. It works by a process of greedy selection and backtracking (if it improves the criterion evaluation). SFFS is computationally feasible for larger feature sets than optimal (branch-and-bound) techniques and it can be used with nonmonotonic criteria (when adding another feature does not always increase the criterion evaluation). Compared with other sub-optimal (plus-l-takeaway-r) techniques, SFFS determines when to backtrack dynamically, without the need for manual parameter setting. It can make multiple sweeps through the feature set to improve performance, but if performance cannot be improved no backward steps are made. In practice, SFFS achieves optimal or near-optimal results [134]. I use SFFS to select efficient subsets of subdomains for test generation.

Algorithm 6 outlines the process used by the SFFS technique. Initially, none of the subdomains are selected. Then, one at a time, subdomains are chosen that most improve the criterion evaluation (mutation adequacy). After a subdomain is added, other subdomains are removed as long as the resulting subsets improve the criterion evaluation at that level. This contrasts with other subset selection techniques (e.g. plus l take away r) that only allow a fixed amount of backtracking.

Algorithm 6 Sequential Floating Forward Selection

1: $k = 0$; $S_0 = \{\}$; $S_N = \{all\_identified\_subdomains\}$
2: **while** $k < desired\_subset\_size$ **do**
3: **Maximise** $J(S_k + s^+)$, **where** $s^+ \in S_N - S_k$
4: $S_{k+1} = S_k + s^+$; $k = k + 1$
5: **Maximise** $J(S_k - s^-)$, **where** $s^- \in S_k$
6: **if** $J(S_k - s^-) > J(S_{k-1})$ **then**
7: $S_{k-1} = S_k - s^-$; $k = k - 1$
8: **end if**
9: **end while**

4.3.2 Subdomain Set Selection Using SFFS

I apply Sequential Floating Forward Selection (SFFS) to identify an efficient selection of subdomain sets that is as small as possible, but still achieves a high level of mutation adequacy. To measure the extent to which a selection achieves these goals, I use the fitness criterion function shown in Equation 4.2.

$$J(S) = \sum_{m \in M} \mathbf{max}_{s \in S}(killed(s, m)) \tag{4.2}$$

(M is the set of mutants, S is the set of subdomains,
$killed(s, m)$ is the number of times subdomain s kills mutant m)

The fitness criterion function is used as follows: first, 100 test cases are sampled randomly from within the bounds of each set of subdomains. This establishes the ability of each set to kill particular mutants. Then, at every step of the selection process, the current selection of subdomain sets is evaluated according to the sum (for each mutant) of the maximum number of times a mutant is killed by any of the included sets. In this way, the criterion function seeks to find a small selection of subdomain sets that kill all the mutants as frequently as possible.

Subdomain set selection starts by finding one subdomain that achieves the highest fitness criterion evaluation on its own, then two subdomains that together form an optimal selection and so on until all the subdomains have been selected. At each step in the process, sets of subdomains may be removed if they improve the fitness criterion evaluation of the smaller selection size. It is with this backtracking capability that SFFS differs from a purely greedy heuristic search. Backtracking allows my technique to improve its selection by taking into account the overlap between mutants killed by different sets of subdomains.

One of the benefits of using SFFS is that it identifies the best sets of subdomains to include for each selection size before moving on to the next one. Once a selection has been confirmed and backtracking has been completed, the algorithm will never go back and change those sets of subdomains. By evaluating the mutation score achieved by each selection size, the point can be found at which adding another set of subdomains will not increase the mutation score any further. My technique is therefore suitable for removing redundant sets of subdomains and finding the smallest selection from which test cases can be sampled without having a detrimental effect on fault finding ability.

4.3.3 Experiments for RQ6

I set up experiments to answer the following research question in regard to selecting sets of subdomains for mutation adequacy:

RQ6: Is it possible to reduce the number of sets of subdomains without significantly affecting the mutation score?

Previously, subdomain optimisation was applied to six Java programs (TriTyp, Schedule, TCAS, Replace, SVD and Schur). Now, Sequential Floating Forward Selection (SFFS) is applied to each set of subdomains identified previously. The previous experiments were conducted with 100 trials. This allows an average to be produced for SFFS. Each selection size is evaluated, starting with a single set of subdomains, and continuing up to every set. At each step, the set of subdomains that achieves the highest criterion evaluation is selected. I record the minimum, maximum and average mutation score for each program and selection size.

Removing sets of subdomains without significantly affecting the mutation score improves fault finding efficiency. I investigate the proportion of the original mutation score that can be retained for various selection sizes. As an example of what is possible with my approach to subdomain set selection, Table 4.5 presents the number of sets of subdomains evolved for each program, along with the reduction that can be achieved without reducing the mutation score by more than 1% of its original value. This is three times less than the reported reduction in mutation score when using expression mutation, upon which the MuJava operators are based [118]. For most programs, SFFS is able to significantly reduce the evolved sets of subdomains with a minimal difference in mutation score.

Table 4.5: Selection of Subdomain Sets Using SFFS (Averaged over 100 Trials)

Program	Original sets	Reduced sets	Reduction	Mutation score
TriTyp	27	11	59.3%	0.991
Schedule	8	7	12.5%	0.923
TCAS	41	17	58.5%	0.774
Replace	91	9	90.1%	0.562
SVD	25	24	4.0%	0.598
Schur	9	4	55.6%	0.915

4.3.4 Results for RQ6

I evaluated the effect that reducing the number of sets of subdomains had on the mutation score for each program. Figure 4.6 and Table 4.6 present the results of selecting every possible number of sets of subdomains, from a single set all the way up to include every set identified by my optimisation process. It is possible to select fewer sets of subdomains with minimal decrease in mutation score for all the programs, except SVD and Schedule. For example, selecting a quarter of the sets of subdomains of TCAS only reduces the mutation score by 3.6% of that achieved using all the sets. Over all, therefore, my approach is highly successful at improving fault finding efficiency by reducing the number of sets of subdomains.

Both SVD and Schedule show a similar trend in that removing sets of subdomains almost immediately decreases the mutation score. After Schur, Schedule and SVD had the smallest number of sets of subdomains. This limits the opportunity for redundant sets of subdomains and makes it more likely for reducing the number of subdomain sets to have a significant effect on the mutation score. The reason why this is not the case for Schur may be because its mutants are easy to kill even by random testing. These negative results do not mean that it is impossible to improve the fault finding efficiency for Schedule and SVD, but that subdomain selection is not the best way to do this.

The mutation scores for Schedule decrease and then increase again as sets of subdomains are removed. This is a little confusing, since removing a set of subdomains cannot increase the criterion evaluation. The reason for this is that the results are averaged over 100 trials and in each trial a different number of subdomains was initially evolved. Many of the graphs are not completely smooth because each optimisation run identified a different number of subdomains.

In the case of Schedule, some of the trials in which a smaller number of sets were evolved, achieved a similar mutation score to trials with larger sets of subdomains. Therefore, even though removing sets of subdomains that have been evolved for schedule nearly always reduces the mutation score, it is possible to evolve fewer sets of subdomains and still achieve a similar mutation score. This suggests that evolving sets of subdomains and then selecting them is not the best strategy for this program. Chapter 5 uses static analysis to provide guidance as to which sets of subdomains to evolve during the optimisation process.

There is a relationship between the sets of subdomains that are selected and the order in which they were identified by the optimisation process. Take for example, the sets of subdomains selected for TCAS in 100 trials (see Figure 4.5). Sets of subdomains are more likely to be included by the selection technique if they were discovered later in the optimisation process. Sets of subdomains that are discovered earlier are more likely to be redundant because they aim to cover mutants more broadly, before some of the mutants have been put aside. This suggests that it is useful to focus on identifying sets of subdomains for harder to kill mutants. A static analysis technique that can provide this guidance may even eliminate the need for subdomain set selection (see Chapter 5).

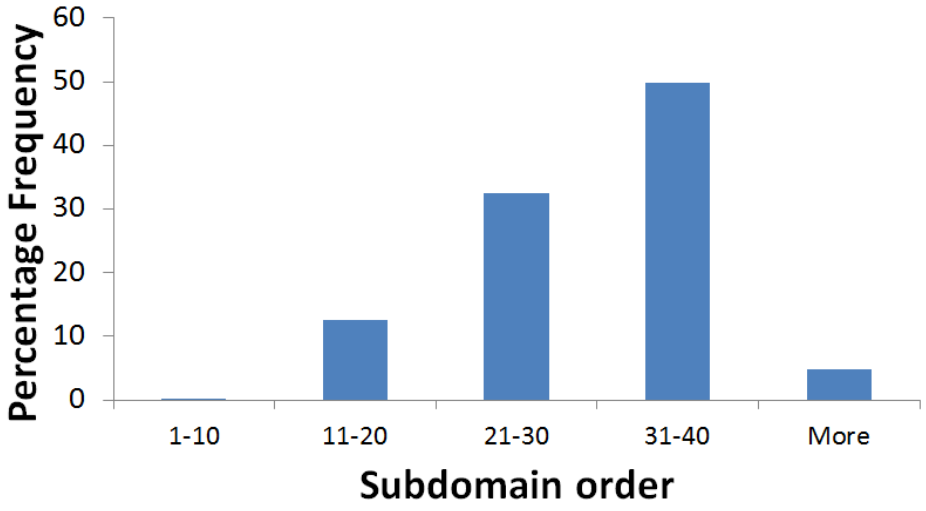


Figure 4.5: Frequency of Subdomains Selected for TCAS Subsets of Size 10

Along with the average mutation score, Figure 4.6 includes the minimum and maximum mutation scores (dotted lines). The minimum mutation score is low for SchurTransformation when selecting the first few sets of subdomains, but this changes quickly after the fifth set is added. It is caused by an optimisation run in which no one set of subdomains has a high mutation score by itself. In general, the minimum, maximum and average values are consistently close to each other, suggesting that the interpretations for RQ6 can be relied upon.

Summary for RQ6: Fewer sets of subdomains can be selected for all but two programs with minimal decrease in mutation score. Subdomains identified later in the optimisation process are more useful than those identified earlier.

Table 4.6: Summary of Results (Averaged over 100 Trials)

Program	Mutation score for selection			
	25%	**50%**	**75%**	**100%**
TriTyp	0.946	0.988	0.994	0.998
Schedule	0.686	0.862	0.828	0.930
TCAS	0.752	0.778	0.779	0.780
Replace	0.523	0.542	0.547	0.566
SVD	0.460	0.531	0.579	0.603
Schur	0.883	0.920	0.921	0.920

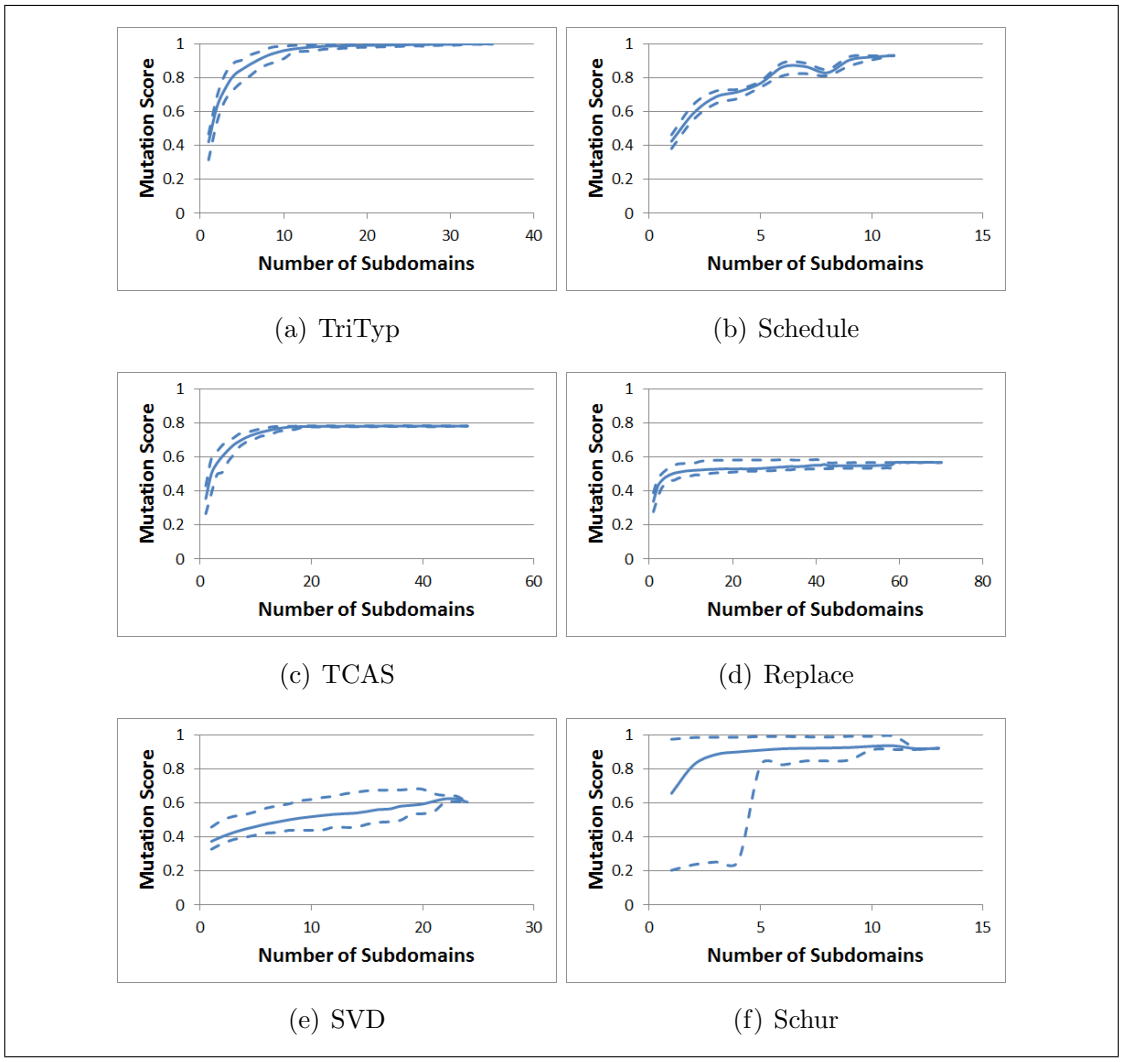


Figure 4.6: Percentage of Mutants Covered by Evolved Subdomains

4.4 Summary

This chapter presented two stages towards producing efficient sets of subdomains. Subdomains are first optimised for their ability to kill mutants consistently, then a small set of subdomains is selected that is able to kill mutants more efficiently. Sampling test cases from multiple optimised subdomains achieved a higher mutation score than single sets of subdomains and random testing, except for one trivially easy to test program. Subdomain selection reduced the number of subdomains for four out of six programs with little effect on mutation score. Multiple sets of subdomains are therefore suitable for some, but not all programs.

Subdomain optimisation has similar computational expense whether single or multiple sets of subdomains are used. Multiple sets of subdomains take less time for some programs and more time for other programs to evolve than single sets. However, multiple sets of subdomains achieve a significantly higher mutation score than single sets (33% on average). In particular, it was previously necessary to manual scale the input parameters of TCAS to achieve an acceptable mutation score. With the new technique for multiple subdomains, a 71% higher mutation score was achieved and parameter scaling was not necessary. Therefore, even for the programs with which multiple sets of subdomains take longer to optimised, the increased mutation score is worth the extra computation.

Subdomain selection significantly improved the efficiency of sampled test suites for the majority of programs. Compared to subdomain optimisation, subdomain selection is computationally inexpensive. It is, however, not effective in all cases: reducing the number of subdomains had an immediate negative effect on Schedule and SingularValueDecomposition. Subdomains identified later in the optimisation process are more useful than those identified earlier. Chapter 5 uses static analysis to guide the targeting of subdomains towards particular mutants such that fewer subdomains need to be removed later.

Chapter 5

Mutant Evaluation by Static Semantic Interpretation

5.1 Introduction

This chapter introduces a new technique for interpreting mutant semantics statically, without executing the program code. Chapter 4 extended the work of Chapter 3 by optimising multiple sets of subdomains, but this introduced redundancy as some of the mutants covered by the evolved subdomains were killed more efficiently by other sets. Semantic interpretation can be used to reduce the number of sets that are evolved without significantly affecting their efficiency.

The idea is that sets of subdomains, targeted at killing mutants semantically similar to the original program, also kill other mutants as a side-effect. Semantically similar mutants can be killed by fewer input values, but they occur on the same paths through the program as other (easier to kill mutants). Subdomains that exercise semantically similar mutants are likely to exercise other mutants.

Semantic similarity can be defined in terms of the mapping of inputs to outputs [117]. Mutants that map all of their inputs to the same output as the original program are said to be semantically equivalent. More formally, mutant (m) is observationally equivalent to the original program (o) if, for all contexts $C[]$, there is a weak bisimulation (R), such that $\langle C[m] \Downarrow, C[o] \Downarrow \rangle \in R$. A set of subdomains is likely to be effective against the complete set of mutants if it is evolved against mutants that are semantically similar, but not equivalent to the original program.

5. MUTANT EVALUATION BY STATIC SEMANTIC INTERPRETATION

Formal methods have been developed to handle approximate similarities [157] and higher order languages [85] in bisimulations. Yet, these techniques require manual involvement and can be computationally expensive [157][85]. I therefore approximate the semantics of mutants in terms of their output range or probability value. Although this does not provide complete accuracy, it can guide subdomains towards mutants semantically similar to the original program.

I determine the extent to which semantic interpretation can be used to select mutants that are similar to the original program using random testing. A mutant is considered to be semantically similar if it is killed by a small number of the test cases sampled at random, but this also depends on the testing methodology employed. In order to minimise this effect, I use a large test suite and a uniform input distribution, but sufficient details are provided for other researchers to repeat the experiments with a different test generation methodology.

Semantic interpretation does not depend on the test generation technique or the size of the test suite. It is no more expensive to evaluate a path through the program that is exercised for a small proportion of inputs as it is to evaluate a path that is exercised for every input. Semantic properties are derived directly from the program code without bias or superfluous executions from a test input distribution. Semantic interpretation provides effective estimates of mutant similarity that can be used to predict how difficult particular mutants are to kill.

There are three main applications for these predictions:

1. **They aid in the identification and removal of equivalent mutants** Equivalent mutants impair upon the veracity of mutation analysis. Semantic interpretation helps the tester focus on mutants that are more likely to be equivalent, thus reducing the time it takes to identify equivalent mutants.

2. **They allow targeting of difficult to kill mutants for test generation** Difficult to kill mutants are more realistic. If they are targeted first, easier to kill mutants are killed as a side-effect. Semantic interpretation identifies difficult to kill mutants for the generation of fewer, more effective test cases.

3. **They point to easy to kill paths for difficult to kill mutants** Difficult to kill mutants can only be killed on a small number of paths through the program. Semantic interpretation can help the tester identify paths that are more likely to reveal a difference in semantics.

5.2 Difference-Based Interpretation

Difference-based interpretation predicts mutants to be semantically similar if their output range for each path resembles that of the original program. Differences are recorded in the minimum and maximum output values of each mutant.

5.2.1 Mutant Semantics

As an example of mutant semantics, let us consider what happens when we mutate an implementation of the remainder operation, as applied to variables x and y (see Algorithm 7). Mutation M3 has no effect on the program output because it post-increments variable *mod* in the print statement, after which it is no longer used. Mutation M1 affects the output of every input, except if x and y are both 1, or if x is 0 and y is not. Mutation M2 only affects the output if *div* is less than zero. It is tempting to conclude that M2 is semantically smaller than M1, but further analysis is required to determine their exact semantic size.

Algorithm 7 Remainder operation

1: $div \leftarrow x/y$	**M1:** $div \leftarrow x * y$
2: **if** $div < 0$ **then**	
3: $mod \leftarrow (div * y) - x$	**M2:** $mod \leftarrow (div * y) + x$
4: **else**	
5: $mod \leftarrow x - (div * y)$	
6: **end if**	
7: print(mod)	**M3: print(mod++)**

Static analysis can be applied to estimate the semantic size of a mutation by inferring (for a given input range) its effect on the output. Calculations are made for each path so as to form predictions about semantics. In this example, mutation M2 is predicted to be smaller than M1 because it affects the output of half the number of paths and only increases the minimum and maximum output values by $2x$ compared to the y^2 factor increase of M1. By contrast, mutation M3 is equivalent to the original program and it has no effect on the output range.

5.2.2 Symbolic Execution

Rather than analyse the semantics of each mutant manually, I have chosen to automate the process of semantic interpretation through the use of symbolic execution. Mutants are selected by comparing their symbolic output expressions with that of the original program. Symbolic execution derives a symbolic output expression for each path through the original program and all of its mutants. The expressions are then fed into a model of arithmetic (see Table 5.1) to determine, for a specific range of input, what the potential range of output will be. Difference-based semantic interpretation compares the minimum and maximum output of each mutant with the minimum and maximum output of the original program.

Figure 5.1 demonstrates the application of symbolic execution to Algorithm 7 in order to help reveal its semantics. The input variables x and y are represented symbolically as X and Y. A new variable (div) is assigned the symbolic expression X/Y and subsequently used in the program. Elsewhere in the diagram, this variable is replaced by its symbolic expression (X/Y). Symbolic execution reveals that the output is ((X/Y)*Y)-X if X/Y is less than zero, otherwise it is X-((X/Y)*Y). The second expression is equal to the first multiplied by -1. This means that the output range (cardinality) is the same for both paths, but the signs and therefore the minimum and maximum values are swapped around (see Table 5.2 for further confirmation of this).

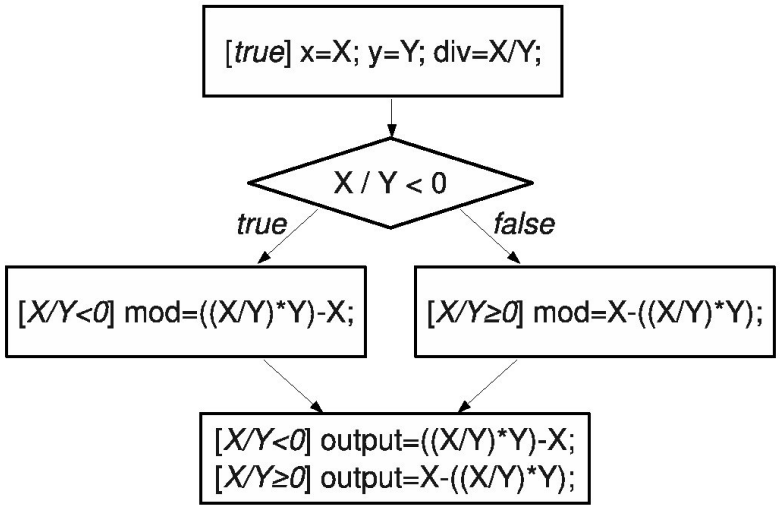


Figure 5.1: Example of Symbolic Execution

5.2.3 Semantic Interpretation

My technique for semantic interpretation applies symbolic execution to each path through the original program and all of its mutants. The resulting symbolic output expressions are used to compute the potential range of output for each path, given a particular range of inputs. Although many of the values in the output ranges cannot be produced by the program, comparing symbolic output expressions in this way allows semantic interpretation to be computationally feasible. The technique does evaluate mutant semantics comprehensively, but it provides enough information to compare their semantic effect.

Algorithm 8 outlines the process of using difference-based semantic interpretation to select mutants that are semantically similar to the original program. A sum of differences (S) is calculated from the minimum and maximum values from the symbolic output expression for each path (p) through a mutant (m) and the original program (o). Mutants are selected according to their similarity with the original program, i.e. in order of their sum of differences (smallest to largest). By sorting the mutants in order of their sum of differences, it is possible to select any number of mutants according to predictions of their semantic similarity.

Algorithm 8 Using semantic interpretation to select mutants

1. Calculate the minimum and maximum output values for each mutant
2. Sum up the size of each difference in minimum and maximum output,

$$S_m = \sum_{p \in Paths} \begin{matrix} |Min(m_p) - Min(o_p)| \\ + \\ |Max(m_p) - Max(o_p)| \end{matrix}$$

3. Remove all the equivalent mutants using random testing and inspection
4. Order the mutants according to their sum of differences
5. Select the mutants most similar to the original program (smallest S_m)

Table 5.1: The Minimum and Maximum Results of Each Operation

	Min(D)	**Max(D)**
L + R	Min(L) + Min(R)	Max(L) + Max(R)
L - R	Min(L) − Max(R)	Max(L) − Min(R)
L * R	if((Min(L) ≥ 0) && (Min(R) ≥ 0)) return Min(L) * Min(R) else if ((Max(L) ≤ 0) && (Max(R) ≤ 0)) return Max(L) * Max(R) else return Smallest(Min(L) * Max(R), Max(L) * Min(R))	if((Min(L) ≥ 0) && (Max(R) ≤ 0)) return Min(L) * Max(R) else if ((Max(L) ≤ 0) && (Min(R) ≥ 0)) return Max(L) * Min(R) else return Biggest(Max(L) * Max(R), Min(L) * Min(R))
L / R	if((Min(L) ≥ 0) && (Min(R) ≥ 0)) return Min(L) / Max(R) else if ((Max(L) ≤ 0) && (Max(R) ≤ 0)) return Max(L) / Min(R) else return Smallest(Max(L) / Smallest(Max(R), -1)), Min(L) / Biggest(Min(R), 1))	if((Min(L) ≥ 0) && (Max(R) ≤ 0)) return Min(L) / Min(R) else if ((Max(L) ≤ 0) && (Min(R) ≥ 0)) return Max(L) / Max(R) else return Biggest(Max(L) / Biggest(Min(R), 1)), Min(L) / Smallest(Max(R), -1))
L % R	if(Min(R) ≥ 0) return 0 else return Biggest(-Max(L), Min(R) + 1)	if(Max(R) ≤ 0) return 0 else return Smallest(Max(L), Max(R) - 1)
-L	-Max(L)	-Min(L)

(Min(X) is the smallest value in X, Max(X) the biggest. Smallest(x,y) is the smallest value of x or y, Biggest(x,y) the biggest)

Minimum and maximum output values are calculated progressively. Each input variable is prescribed a set range; constants have the same minimum and maximum value. Then, an output range is calculated for every arithmetic operation, according to the operators and inputs applied. Table 5.1 lists the minimum and maximum output value for each arithmetic operator in terms of the range of inputs. Calculations for addition and subtraction are the same regardless of the sign of the inputs, but multiplication and division are more complicated. The minimum result of division is Min(L)/Max(R) when all the input values are positive, but Max(L)/Min(R) when all the input values are negative.

The minimum and maximum output values of arithmetic operations are themselves, in turn, applied as the input range for further operations. This procedure is performed repeatedly so that, through a series of sub-expressions, every arithmetic expression has its minimum and maximum output value calculated according to the input range of the program. Ultimately, the output range of every path through the program can be determined from the arithmetic expression in its return statement, as determined by symbolic execution and the range of each input variable. Table 5.2 shows the application of these calculations to Algorithm 7 with input values in the range 1 to 100. The difference sums confirm that mutant M3 is likely to be equivalent because its difference is zero for both paths. M2 is semantically more similar to the original program than M1 because one of its paths has a zero difference and the other has a smaller difference than M1.

Table 5.2: Symbolic Execution for Selective Mutation on Algorithm 7

	Path	**Output**	**Min**	**Max**	**Difference (S_m)**
Original	Left	((X/Y)*Y)-X	-99.99	9900	-
	Right	X-((X/Y)*Y)	-9900	99.99	-
M1	Left	((X*Y)*Y)-X	-99	999999	990099.99
	Right	X-((X*Y)*Y)	-999999	99	990099.99
M2	Left	((X/Y)*Y)+X	1.01	10100	301
	Right	X-((X/Y)*Y)	-9900	99.99	0
M3	Left	((X/Y)*Y)-X	-99.99	9900	0
	Right	X-((X/Y)*Y)	-9900	99.99	0

5.2.4 Experiments for RQ7

Experiments were set up to answer the following research question about mutant selection via difference-based static semantic interpretation:

RQ7: Can difference-based semantic interpretation be used to select mutants that are more difficult to kill than those on average?

RQ7 is addressed by applying difference-based semantic interpretation to select the 10%, 25% and 50% most semantically similar mutants of each method. For each selection size, I record the mutation score of one million random test cases along with the average frequency with which a random test case kills each mutant. These measures can be used together to reveal interesting properties about each sample. A high mutation score and low frequency would suggest that most of the mutants *can* be killed, but they are very difficult to kill (with the test suite employed). A low mutation score and high frequency indicates that some of the mutants are difficult (or impossible) to kill, but some are also very easy.

I consider difference-based interpretation to be successful if it selects mutants that are difficult to kill by random testing. This can be evaluated using the results recorded for mutation score and mutant killing frequency. If mutants are selected that are difficult to kill, then as the sample size decreases, I would expect to see a decrease in the mutation score, the killing frequency or both. If neither of these measures decrease, then difference-based interpretation has not been successful.

5.2.4.1 Test Subject Programs

Difference-based semantic interpretation is applied to mutants generated from various numerical programs. Numerical programs are used because difference-based interpretation can not reason semantically about strings or objects. It is also important to select methods that are sufficiently large so as to avoid calculations involving trivial semantics. These requirements are met by selecting methods according to the following criteria: Their return type must be numeric, they must take at least one parameter, all their parameters must be numeric and they must occupy over 1KB in memory. Nineteen methods were selected from the Java standard library that meet these criteria (see Table 5.3).

Table 5.3: Subject Methods from the Java Standard Library [123]

		LOC	Mutants	Equivalent Mutants
java.math.BigDecimal				
1	$int\_checkScale(long)$	16	60	15
2	$int\_longCompareMagnitude(long, long)$	18	44	15
3	$long\_longMultiplyPowerTen(long, int)$	18	98	28
java.math.BigInteger				
4	$int\_getInt(int)$	18	46	0
javax.swing.JTable				
5	$int\_limit(int, int, int)$	5	36	1
javax.swing.plaf.basic.BasicTabbedPanelUI				
6	$int\_calculateMaxTabHeight(int)$	8	42	2
7	$int\_calculateMaxTabWidth(int)$	8	42	2
8	$int\_calculateTabAreaHeight(int, int, int)$	8	51	1
9	$int\_calculateTabAreaWidth(int, int, int)$	8	51	1
10	$int\_getNextTabIndex(int)$	4	17	0
11	$int\_getNextTabIndexInRun(int, int)$	12	63	18
12	$int\_getPreviousTabIndex(int)$	4	50	18
13	$int\_getPreviousTabIndexInRun(int, int)$	12	91	24
14	$int\_getRunForTab(int, int)$	9	43	6
15	$int\_lastTabInRun(int, int)$	11	81	11
javax.swing.plaf.basic.BasicTreeUI				
16	$int\_findCenteredX(int, int)$	5	49	49
17	$int\_getRowX(int, int)$	3	25	0
javax.swing.text.AsyncBoxView				
18	$float\_getInsetSpan(int)$	5	27	1
19	$float\_getSpanOnAxis(int)$	7	17	1

The Java standard library [123] is a large collection of classes that provide extensions to the core functions of the Java programming language. The chosen methods reside within four classes for user interface coordination (JTable, BasicTabbedPaneUI, BasicTreeUI and AsyncBoxView) and two classes for handling large numerical data types (BigDecimal and BigInteger).

5.2.4.2 JPF-Symbc: Symbolic Execution Tool

Java Pathfinder (JPF) is an open source model checker and Java virtual machine, originally developed by NASA to find concurrency faults [148]. Extensions have been written for JPF to handle a variety of testing and verification tasks. JPF-symbc [133] is a symbolic execution extension for JPF (see Figure 5.2). It performs symbolic execution by storing symbolic attributes along with each variable on the stack. JPF-symbc is capable of processing integer and real numeric values, Booleans, references and strings. A number of constraint solving packages are also included for finding input values to exercise each path. JPF-symbc has been used by Fujitsu to test web applications [52] and has helped find a bug in the Onboard Abort Executive for the NASA Crew Exploration Vehicle [133].

JPF-symbc explores all the paths in the program it is supplied. Therefore, it is important to isolate the method under test from the rest of the program. I pre-process each Java method for analysis with JPF-symbc by extracting it from the class in which it is defined, along with any other methods on which the class depends. I then introduce local constants to replace all calls to retrieve data from outside of the class and remove any calls to output data via a side effect. To avoid path explosion problems, each loop is only allowed to run once. These preprocessing transformations are not semantically preserving, but they allow individual methods to be tested independently from the rest of the program.

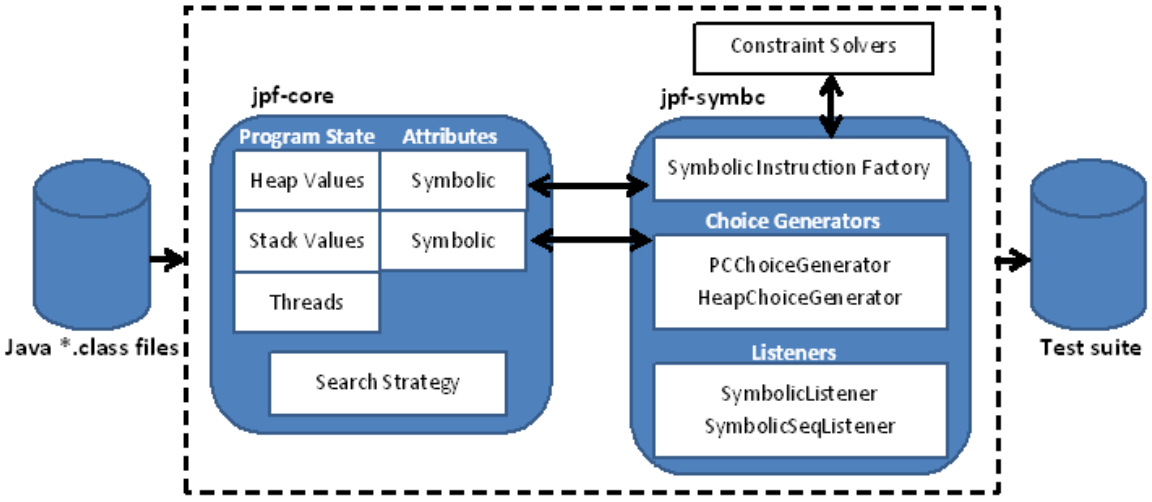


Figure 5.2: JPF-Symbc Extension for Java Path Finder [133]

5.2.5 Results for RQ7

RQ7 is answered by using random testing to evaluate the selections made by semantic interpretation. Table 5.4 shows the mutation score for selections made from each method, along with the average frequency with which a random test case kills each mutant. Difference-based semantic interpretation was able to select difficult to kill mutants for most of methods evaluated from the Java standard library. Of the first ten percent most difficult to kill mutants selected, 42.6% of them were killed, compared to 69.4% of the complete set. The probability of killing a mutant in the first quarter is 20.40%, compared to 38.37% in the complete set. This means that the average mutant in the top quarter of those selected by semantic interpretation is less than half as likely to be killed by a random test case than the average mutant in the remaining three quarters.

There are, however, a number of cases where mutation score and killing frequency does not decrease as a result of difference-based selection. In some cases, this is due to the way the results are collected and presented. For example, Methods 13, 16 and 19 have killing frequencies of 0.00% for certain selection sizes - the values are actually slightly greater than zero, but rounding prevents any further decreases in killing frequency from being reported. No decrease in mutation score is observed for any selection size of Method 6, as all of its mutants are killed by the random test suite - a significant decrease in killing frequency indicates that semantic interpretation is still able to select mutants that are difficult to kill.

In other cases, difference-based selection does not reduce the mutation score or killing frequency due to weaknesses in the technique. For example, the lowest frequency with which mutants were killed from Method 15 occurred when using the complete set - all selections resulted in a significantly higher killing frequency. The mutation score of Method 4 decreases to 0.500 when 50% of the mutants are selected, but then increases to 0.527 for 25% before returning to 0.500 again when 10% of the mutants are selected. It can therefore be said that, although difference-based interpretation *does* identify more difficult to kill mutants, there are specific instances where it fails to achieve this goal. One reason for this is that differences in the range of symbolic output expressions do not take into account the output distribution or branch structure. They are only an approximation of the likelihood that mutants produce the same output as the original program.

Table 5.4: Results for 19 Methods from the Java Standard Library

	Mutation score				Frequency killed by random test case			
	10%	**25%**	**50%**	**100%**	**10%**	**25%**	**50%**	**100%**
1:	0.725	0.739	0.752	0.822	22.67%	24.72%	22.90%	29.17%
2:	0.817	0.829	0.812	0.828	12.50%	13.33%	11.90%	13.79%
3:	0.267	0.239	0.404	0.671	5.95%	6.07%	4.29%	8.57%
4:	0.500	0.527	0.500	0.609	13.80%	14.15%	12.99%	13.72%
5:	0.522	0.504	0.647	0.829	26.29%	28.68%	24.51%	32.50%
6:	1.000	1.000	1.000	1.000	59.05%	60.11%	69.36%	84.11%
7:	0.442	0.433	0.550	0.775	26.08%	26.05%	38.15%	65.18%
8:	0.447	0.456	0.720	0.860	37.86%	37.33%	67.29%	82.34%
9:	0.493	0.486	0.720	0.863	41.18%	40.62%	67.28%	82.66%
10:	0.733	0.658	0.658	0.765	73.33%	65.83%	65.83%	72.06%
11:	0.175	0.200	0.227	0.444	4.79%	5.45%	5.72%	13.89%
12:	0.211	0.229	0.563	0.788	14.44%	17.08%	26.30%	45.45%
13:	0.033	0.031	0.023	0.162	0.00%*	0.00%*	0.00%*	3.68%
14:	0.500	0.489	0.500	0.514	7.22%	9.6%	9.45%	12.16%
15:	0.644	0.618	0.599	0.710	11.94%	14.31%	13.82%	10.15%
16:	0.000	0.000	0.208	0.612	0.00%	0.00%	7.85%	28.32%
17:	0.333	0.378	0.500	0.760	33.33%	37.78%	50.00%	75.50%
18:	0.633	0.611	0.641	0.615	25.00%	19.72%	22.56%	34.62%
19:	0.267	0.225	0.250	0.563	0.00%*	0.00%*	0.00%*	31.25%
Mean:	**0.426**	**0.455**	**0.541**	**0.694**	**21.86%**	**21.40%**	**26.65%**	**38.37%**

*Rounded down to zero from a slightly higher number

Figures 5.3 and 5.5 present these results graphically so that the general trends in mutation scores and decreased mutant killing frequencies can be seen Smaller selection sizes typically result in lower mutation scores and decreased mutant killing frequencies. However, for some methods, the mutation scores remain largely unaffected by selection size (see Figure 5.3). It seems therefore that difference based interpretation does not work equally as well on all programs.

Many of the methods that show a small decrease in mutation score for smaller selection sizes also show a small decrease in mutant killing frequency (see Figure 5.5). There are, however, more outliers in the killing frequency results. Method 16 and 19 show a sharp decrease in killing frequency as the selection size is reduced, Mutant 10 and 14 show a much more gradual decrease and Mutant 15 actually shows an increase in frequency. As discussed previously, if a method has a steeper gradient for killing frequency than mutation score, this suggests that more difficult to kill mutants are being identified even when there is no clear distinction in mutation score. If, however, a method has a steeper gradient for mutation score, some of the mutants selected may be overly easy to kill.

Difference-based interpretation performs well as an approximate measure of semantic similarity (see Figures 5.4 and 5.6). There is a difference of 11.5% in mutant killing frequency between selecting all of the mutants and the half that are predicted to be the most similar. In contrast, the difference between selecting half of the mutants and one quarter is just 5.23% - as a result of the large amount of variance in these results, Wilcoxon and students T-tests reveal this difference to be insignificant (see Table 5.6). There is a statistical significance in making this selection in terms of mutation score (see Table 5.5), yet the difference between selecting 25% and 10% of the mutants is still insignificant. Difference-based selection can be used successfully to select the half or maybe quarter of mutants that are the most difficult to kill, yet to achieve more accurate results than this, it is necessary to consider more advanced forms of semantic similarity measurement.

Summary for RQ7: Difference-based interpretation can select difficult to kill mutants for most methods evaluated from the Java standard library. Mutants in the top quarter of those selected were less than half as likely to be killed than the average mutant in the remaining three quarters. Nevertheless, difference-based interpretation does not have a high correlation for all methods. The next section addresses this problem with a new technique for probability-based selection.

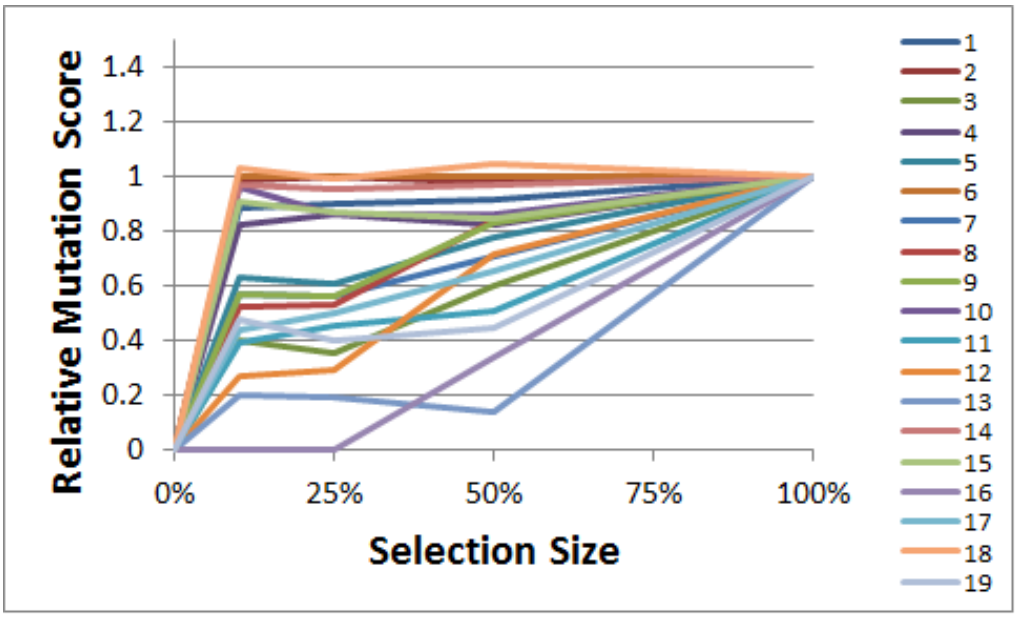


Figure 5.3: Relative Mutation Score of Selected Mutants

Table 5.5: Statistical Analysis of Mutation Score

	100%-50%	**50%-25%**	**25%-10%**
Mean	0.153	0.085	-0.005
Variance	0.013	0.012	0.001
T-Value	5.853	3.355	-0.751
Critical	2.552 **(reject)**	2.552 **(reject)**	2.552 **(accept)**
Wilcoxon	168+ 3-	135.5+ 17.5-	66+ 87-
Critical	32 **(reject)**	27 **(reject)**	27 **(accept)**

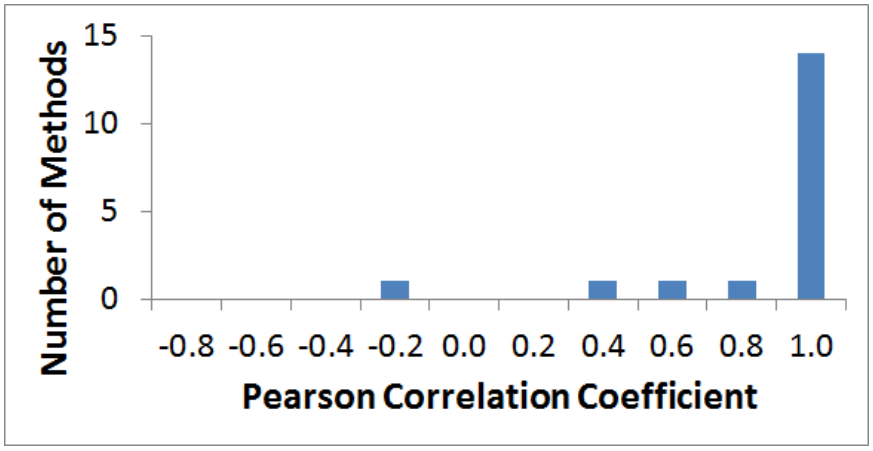


Figure 5.4: Correlation between Mutation Score and Selection Size

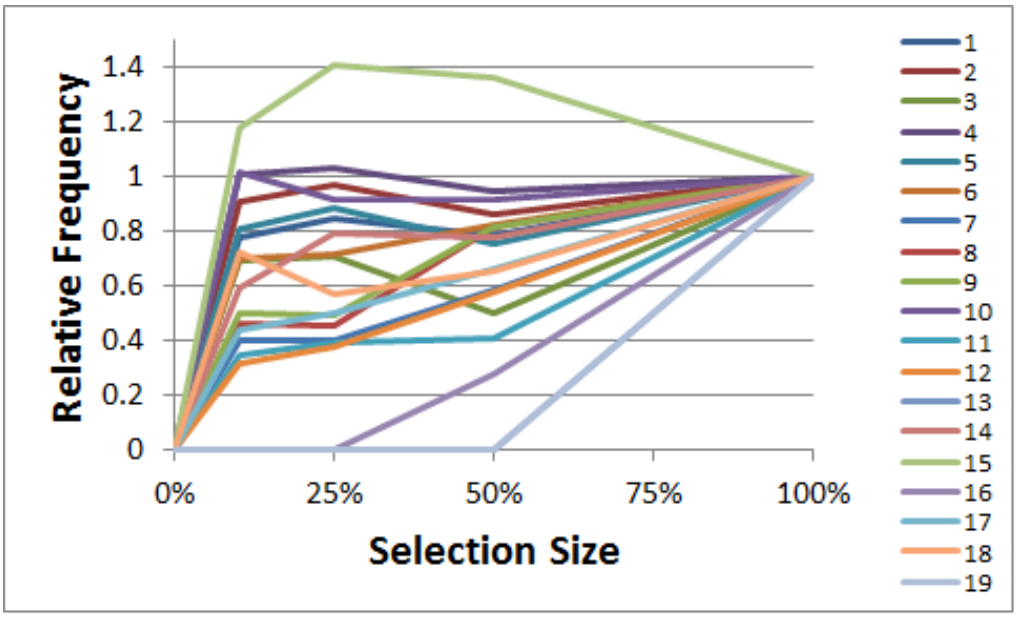


Figure 5.5: Relative Killing Frequency of Selected Mutants

Table 5.6: Statistical Analysis of Frequency

	100%-50%	**50%-25%**	**25%-10%**
Mean	0.115	0.0523	0.0286
Variance	0.975	0.958	0.272
T-Value	5.149	2.378	0.458
Critical	2.552 **(reject)**	2.552 **(accept)**	2.552 **(accept)**
Wilcoxon	185+ 5-	130+ 60-	117+ 53-
Critical	37 **(reject)**	37 **(accept)**	32 **(accept)**

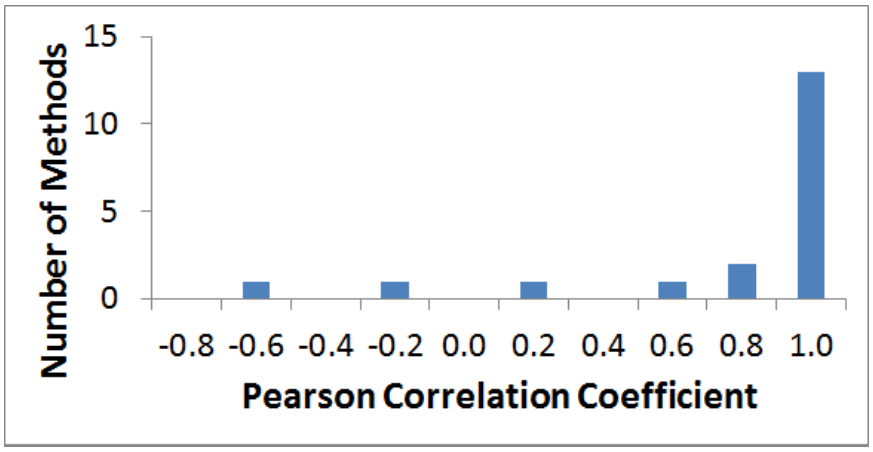


Figure 5.6: Correlation between Killing Frequency and Selection Size

5.3 Probability-Based Interpretation

Probability-based interpretation predicts mutants to be semantically similar to the original program if there is a high probability they will produce the same output. It takes into account the proportion of output values that are the same as the original program, rather than their numerical difference. The previous (difference-based) metric can only make assessments of semantic similarity for numerical data types (i.e. integers or floating point values). Probability-based interpretation can also be used as a practical tool for interpreting the semantic similarity of mutants that involve Booleans, bit vectors, strings and objects.

Probability-based interpretation is more accurate than difference-based interpretation. Take for example two mutants, one with an output range of [1,10] and the other [1,100]. If the output range of the original program is [11,20], the sum of differences for the first mutant is 20 and the second 90. The difference-based metric incorrectly assumes that the first mutant is more similar to the original program because its minimum and maximum values are numerically closer. Yet, it is actually impossible for the first mutant to produce the same output as the original program. Probability-based interpretation reveals that the second mutant is more similar because it has 0.1 probability of producing the same output as the original program, compared to the zero probability of the first mutant.

5.3.1 Boolean Expressions

Boolean expressions can be represented numerically with 1 (for true) or 0 (for false). Limiting their range to just two values, however, provides little semantic information. Only operations involving tautologies or contradictions have any effect on the comparison of their outputs. I represent Boolean expressions in terms of their probability of being true (see Table 5.7). This allows more accurate estimation of the effect of each mutation in semantic output comparisons.

Boolean input values are assumed to be independent of each other and their probability of being true is set to 0.5. This approximation allows similarity assessments to be made without prior knowledge of the input distribution. If an input variable is used more than once in an output expression, its probability value is replaced by 1 (true) or 0 (false) after the first use, to avoid taking its probability into account more than once. For example $x \&\& x = x \&\& true = x$

Table 5.7: Semantic Interpretation of Boolean Operations

Operation	Probability Output True
$P(L\&\&R = true)$ (conjunction)	$P(L = true) * P(R = true)$
$P(L \mid\mid R = true)$ (disjunction)	$P(L = true) + P(R = true) - P(L\&\&R = true)$
$P(L \wedge R = true)$ (exclusive-or)	$P(L \mid\mid R = true) - P(L\&\&R = true)$

and $x \mid\mid x = x \mid\mid false = x$. I measure the probability that two Boolean expressions have the same output value by considering the probability that both expressions are true or that both expressions are false (see Equation 5.1).

$$P(m = o) = P(m = true) * P(o = true) + P(m = false) * P(o = false) \quad (5.1)$$

5.3.2 Bit Vectors

Although bit vectors are stored in numerical data types, bit operations act upon the value of each individual bit (1 or 0), rather than the numerical value as a whole. Bit operations were previously modelled in terms of their output range (see Table 5.1). Yet, it is more accurate to consider the effect they have at the bit level. The probability of a bit value being 1 in the output of a conjunction, disjunction or exclusive-or can be determined by the same model as was used for Boolean values (where $1 = true$ and $0 = false$). Equation 5.2 gives the probability of two bit vector expressions returning the same output value.

$$P(m = o) = \prod_{i=0}^{n} P(m_i = o_i), \text{where n is most significant bit set in } o \text{ or } m \quad (5.2)$$

Bit vectors can also be included as terms of numerical operations (plus, minus, multiply etc.). These operations are modelled by transforming the probability of each bit being true into a minimum and maximum value for the bit vector (see Equation 5.3). The minimum value of a bit vector is determined by the bits fixed at 1 and the maximum value by the largest significant bit and the bits fixed at 0.

$$Min(x) = \sum_{i=0}^{n} [P(x_i{=}1){=}1] * 2^i \qquad Max(x) = 2^{n+1} - 1 - \sum_{i=0}^{n} [P(x_i{=}1){=}0] * 2^i \quad (5.3)$$

$$\text{Iverson bracket notation: } [P] = \begin{cases} 1 \text{ if } P \text{ is true} \\ 0 \text{ otherwise} \end{cases}$$

5.3.3 Numerical Expressions

The semantic similarity of two numerical expressions is predicted in terms of the probability that values sampled at random from within their output ranges will be the same. This can be calculated by considering the number of integer values in each output range that are the same (see Equation 5.4). To simplify this calculation, numerical expressions are assumed to produce a uniform distribution.

$$P(m = o) = \frac{|m^* \cap o^*|}{|m^*| * |o^*|} \tag{5.4}$$

(m^*_p is the range of outputs from mutant m along path p)

Numerical expressions may be used within a Boolean expression, as part of an equality (e.g. $x == y$) or an inequality (e.g. $x > 5$). The probability that an equality is true can be predicted using Equation 5.4, but inequalities require a new prediction method (see Equation 5.5). The calculation is similar to that for equalities, except that instead of counting the number of values that appear in both ranges, it is necessary to count the number of values in the second range that are smaller than each value in the first. Equation 5.5 describes the calculation for a greater-than operation, but it is trivial to swap the ranges around for a less-than operation or the probability that $P(a > b)$ is not true for less-than-or-equal-to.

$$P(a{>}b) = \frac{\begin{array}{c}[\max(a^*) - \max(b^*)] * |b^*| + |a^* \cap b^*| \\ * \min(\max(a^*), \max(b^*)) - \min(b^*) - k)/2\end{array}}{|m^*| * |o^*|} \tag{5.5}$$

($k = 1$ for discrete comparisons, $k = 0$ for continuous)

The probability that a numerical inequality is satisfied influences the probability that other inequalities in the expression are satisfied. Yet, unlike pure Boolean expressions, dependencies between inequalities cannot be eliminated simply by the removal of repeated terms. In the simplest case, dependencies can be identified as overlapping regions (e.g. $x{>}5 \&\& x{>}10 \rightarrow x > 10$). More complex dependencies require the inequalities to be rewritten (e.g. $x{+}y{>}3 \&\& x{+}2y{>}6 \rightarrow y{>}3$).

I handle dependencies between inequalities using the simplex algorithm for linear algebra [38]. Inequalities are simplified and removed through a process of Gaussian elimination. Linear approximation is applied for any non-linear expressions. I prepare Boolean expressions for the simplex algorithm by rewriting them in canonical form (disjunction of conjunctions) with De Morgan's laws. The simplex algorithm is applied to simplify each conjunction separately, before simplifying the disjunction as a whole using the rule $P(x \mid\mid y) = P(x) + P(y) - P(x \&\& y)$.

5.3.4 String Operations

I model string processing on a character-by-character basis with a combination of symbolic and concrete characters. Symbolic characters represent unknown values from a given set (e.g. the basic Latin alphabet) that will not be determined until the program is executed. Concrete characters, on the other hand, have known values that are determined statically in advance by the program code. Each input string is represented as an array of symbolic characters. As well as extracting and reordering characters, string operations can also introduce new concrete characters. By keeping track of the individual characters, it is straightforward to determine the effect of each string operation on the output (see Table 5.8).

Table 5.8: Semantic Interpretation of Popular String Operations

Operation	**Output**
concat(a, b)	$a_1a_2a_3a_4a_5..a_{\|a\|}b_1b_2b_3b_4b_5..b_{\|b\|}$
equals(a, b)	$[\|a\| = \|b\|] \prod_{i \in \|a\|} ([a_i \equiv b_i] + [a_i \not\equiv b_i \& (a_i \in S \| b_i \in S)]/N)$
length(a)	$\|a\|$
substring(a, b, c)	$a_b..a_c$
startsWith(a, b)	$[\|a\| >= \|b\|] \prod_{i \in \|b\|} ([a_i \equiv b_i] + [a_i \not\equiv b_i \& (a_i \in S \| b_i \in S)]/N)$

(S is the set of symbolic inputs, N is the number of values assigned to each input (a..z would be 26) and $a \equiv b$ is true if a and b are identical symbolic or concrete values)

In a survey of 38 popular Java applications, concatenation accounted for 68% of all string operations [137]. By modelling just the top 5 most commonly used operators it is possible to account for 90% of string processing (see Table 5.8). Many of the operators in the remaining 10% can be rewritten in terms of these 5 basic operators, but if an unexpected operator is encountered it can be ignored. Some operators (concat, length and substring) can be performed on symbolic or concrete characters without having to reason about their value. The other operators (equals and startsWith) can be modelled probabilistically. The probability of a symbolic and concrete character or two symbolic characters having the same value is $1/N$, where N is the size of the character set. If there are different symbolic or concrete characters at any point, or the length of the input strings is incompatible, these operations have a zero probability of being true. Otherwise the probability is estimated on the basis of independent probability theory.

5.3.5 Objects

Ciupa et al. [33] describe a metric for measuring the distance between two objects in adaptive random testing. Their metric takes into account differences in the objects' type, their field values and the field values of any other objects they reference. I have adapted this metric to be used for static semantic analysis in calculating the probability that a mutant and the original program have the same output value (see Equation 5.6).

$$P(m = o) = \begin{cases} \prod_{i \in fields(o)} P(m.i = o.i) & \text{if } type(m) = type(o), \\ 0 & \text{otherwise} \end{cases} \tag{5.6}$$

I use a recursive process to calculate the probability that two objects have the same value. Each object contains a collection of fields, which in turn may have primitive types or references to other objects. Two objects cannot have the same value if their primitive fields have a different type or they refer to differently typed objects. In this case, the probability of their equivalence is zero.

Any pair of objects that have the same type can be compared recursively to assess their semantic similarity. I compare all the primitive field values inside these objects along with those contained in objects they refer to. Primitive values are compared symbolically, using symbolic expressions expressed in terms of the program input values. Metrics have been described previously in this section to estimate, for a variety of primitive types, the probability that two symbolic expressions output the same value.

5.3.6 Control Flow Probability

The probability that a mutant produces the same output value as the original program can be estimated by taking the sum of probabilities that the output is the same along each path (see Equation 5.7). This value has a similar role to the sum of differences from the difference-based metric. As such, it assumes each path has the same probability of being executed. Yet this can have a significant effect when certain paths are more likely to be executed than others.

$$P(m = o) = \sum_{p \in Paths} P(m_p = o_p) \tag{5.7}$$

Probability-based interpretation provides a way to predict the likelihood that particular paths through a program are exercised. This is important for two reasons: Firstly, it is necessary for interpreting the semantic effect of mutations made directly on branch conditions; secondly, semantic differences have a greater effect on the output if they occur on frequently exercised paths. Modelling a program's control flow allows more accurate semantic interpretation of its mutants.

I estimate path execution frequency by considering all the branch conditions along each path. Program code mutations can affect the control flow in two ways: directly, by changing the operations applied in a branch condition; or indirectly, by changing the value of a variable that is later used in a branch condition. I have extended the semantic interpretation heuristic to take control flow into account by calculating for each mutant the likelihood that each of its paths are executed.

I estimate the probability of satisfying each branch condition using the metrics previously described for Boolean probabilities. For example, branch condition $x == 1$ has probability 0.01 of being exercised if the input domain of x is $[0, 99]$. I calculate the probability of satisfying each branch condition, under the assumption that all the probabilities are independent and a fixed input domain is used for test data generation. As before, repeated terms and overlapping conditions are removed in an attempt to reduce the number of dependencies. The resulting metric is still not completely realistic, but it allows reasonable approximations to be made within a computationally feasible time frame.

Equation 5.8 extends the sum of probabilities calculation for control flow. There is an additional term for the probability that the mutant and the original program execute different paths. This is calculated by collecting the constraints together, then eliminating dependencies and repeated expressions. In addition, the probability of producing the same output value for each path has been augmented with the probability that path is executed. These changes were made to provide semantic analysis for mutations made directly to branch conditions and address the effect that changes in control flow have on path execution frequency.

$$P(m = o) = \sum_{p \in Paths} P(o_p) - P(m_p \&\& o_p) + P(m_p \&\& o_p) * P(m_p = o_p) \qquad (5.8)$$

($P(o_p)$ is the probability that the original program executes path p,
$P(m = o)$ that the mutant has the same output as the original program)

5.3.7 Example

As an example of probability-based semantic interpretation, Algorithm 9 implements an ISBN class in Java. The class constructor inputs ISBN prefix, group, publisher and book codes, checks they are within the range of the ISBN standard, then sets the value of the ISBN code accordingly. Mutants may produce a different output to the original program if they change the validation conditions, alter the circumstances under which a zero is added to the publisher code or calculate the check digit differently. Three mutations are annotated on the program code: M1 replaces a less-than sign with a greater-than sign in the validation conditions; M2 moves the zero addition threshold from 10 to 5; M3 replaces a minus sign in the check digit calculation with a plus. These mutations demonstrate various semantic changes probability-based interpretation must take into account.

Algorithm 9 Example ISBN class written in Java

```
1  public class ISBN {
2     public String code = ""; public boolean valid = false;
3     public int get(int pos)
4        { return Integer.parseInt(code.substring(pos, pos)); }
5     public ISBN(int prefix, int group, int publisher, int book)
6     {
7        if ((prefix<* 978)||(prefix>979)||(group<0)||(group>99999)||
             (publisher<0)||(publisher>9999999)||(book<0)||(book>999999))
8              { valid = false; }
9        else
10       {
11          code+=prefix+group+((publisher<10†)?"0":"")+publisher+book;
12          valid = (code.length() == 12);
13          if (valid)
14          {
15             int sum=(get(0)+3*get(1)+get(2)+3*get(3)+get(4)+3*get(5)+
                   get(6)+3*get(7)+get(8)+3*get(9)+get(10)+3*get(11));
16             code += ((sum%10==0)?"0":(10 −‡ sum%10));
17          }
18       }
19    }
20 }
```

* M1 changes < into > † M2 changes 10 into 5 ‡ M3 changes − into +

Mutation M1 affects the output of Algorithm 9 by changing one of its branch conditions. It is therefore necessary to know the probability that this change will have an effect on the execution path of the resulting mutant. Equation 5.8 determines this by exploring all the paths through the original program. M1 will cause the program to execute a different path if the mutated branch condition evaluates true in the mutant, but not the original program, or vice versa. This occurs when the value of prefix is not equal to 978, so assuming an input range of [0,999], the probability that the mutant will execute a different path is $977/1000 + 22/1000 = 0.999$. Since M1 occurs at the beginning of the ISBN constructor, the probability the mutation is exercised is 1. Therefore, using Equation 5.8, the probability that the mutant produces a different output is also 0.999.

The probability that M2 has an effect on the output is calculated in a similar way. The probability that *publisher* is less than 10 but not less than 5 is equal to $(10 - 5)/1000$ or 0.005. The probability that *publisher* is less than 5 but not less than 10 is equal to *zero*. M2 will only be exercised if the first branch condition evaluates false. The probability of this is $(21 * 980)/(1000^2)$, assuming an input range of [1,1000] for each parameter. The probability that M2 has an effect on the output is therefore $0.02058 * 0.005 = 1.029e - 4$.

The probability that M3 has an effect on the output is calculated by considering the likelihood that it is executed and changes the value of *code*. M3 is exercised if the first branch evaluates false ($P = 0.02058$), the length of *code* is 12 ($P = 0.9^4$) and $sum\%10$ is not equal to zero ($P = 0.9$). M3 just affects the check digit. In the original program, its output range is $[10, 10] - [0, 9] = [1, 10]$. In the mutant, this becomes $[10, 10] + [0, 9] = [10, 19]$. These ranges only overlap once (with 10). The probability that M3 has an effect on the output is therefore $0.02058 * 0.9^5 * 0.99 = 0.012031$. Assuming every input parameter has the range $[0, 999]$, M1 is predicted to have the greatest semantic effect and M2 the smallest.

5.3.8 Experiments for RQ8

Experiments were set up to answer the following research questions about mutant selection via probability-based static semantic interpretation:

RQ8: Can probability-based semantic interpretation be used to select mutants that are on average more difficult to kill than those selected by difference-based semantic interpretation?
Probability-based interpretation should be more effective than difference-based interpretation because it provides more accurate estimates of the likelihood that mutant outputs have the same value as the original program. I address RQ8 by selecting mutants according to both predictions of semantic similarity, then evaluate the selections with one million random test cases. Probability-based interpretation can be considered successful if identifies mutants that are more difficult to kill on average than difference-based interpretation.

RQ9: Does probability-based semantic interpretation reduce the number of subdomains that can be removed after optimisation without affecting their mutant-killing effectiveness?
Semantic interpretation aims to identify mutants that are difficult to kill. When optimising multiple sets of subdomains (see Chapter 4), difficult to kill mutants are typically covered late in the optimisation process. These subdomains frequently also kill mutants that were covered earlier. As a result, it is often possible to remove subdomains identified early in the optimisation process without affecting the mutation score of test suites that are sampled from them. It is computationally inefficient to evolve subdomains, only to remove them later. RQ9 investigates whether evolving subdomains only for the mutants identified by semantic interpretation as being difficult to kill reduces the number of subdomains that can be removed without affecting the average mutation score. I address this research question by optimising multiple sets of subdomains for each selection size of mutants. I record the mutation scores achieved by the optimised subdomains, along with the number of subdomains that can be removed once subdomain optimisation is completed without affecting the mutation score.

5.3.9 Test Subject Programs

The probability and difference-based interpretation techniques were applied to mutants generated from three Java programs (see Table 5.9). These programs are larger than those previously used to evaluate difference-based interpretation. Their increased complexity should make semantic analysis more challenging.

Table 5.9: Benchmark Programs From Testing Research

Program	**Mutants**	**LOC**	**Reference**
FourBalls	189	40	[147]
TriTyp	310	61	[147]
TCAS	267	120	[128]

5.3.10 Results for RQ8

Table 5.10 shows the mutation score for selections made from FourBalls, TriTyp and TCAS using difference-based and probability-based interpretation. Both metrics can be used to select difficult to kill mutants, but with different degrees of success. When selecting 10% of the mutants, difference-based interpretation produces an average mutation score of 0.267, whereas probability-based selection produces an average mutation score of 0.185.

Table 5.10: Mutation Score Results

Programs	Difference-based metric				Probability-based metric			
	10%	**25%**	**50%**	**100%**	**10%**	**25%**	**50%**	**100%**
FourBalls	0.434	0.423	0.492	0.463	0.345	0.321	0.324	0.463
TriTyp	0.274	0.225	0.235	0.267	0.155	0.206	0.223	0.267
TCAS	0.094	0.115	0.093	0.097	0.054	0.063	0.105	0.097
Mean	**0.267**	**0.254**	**0.273**	**0.276**	**0.185**	**0.197**	**0.217**	**0.276**

Table 5.11 shows the average frequency with which a random test case kills each mutant. At first sight, the difference between the two selection techniques seems quite small. Difference-based interpretation reduces the killing frequency for a 10% selection down to 14.5%, whereas probability-based selection reduces it to 12.5%. Yet, it is important to read these results in light of the others. For 25% selections, the difference-based metric achieves 18.0% killing frequency compared to 12.6% with probability-based selection. Overall, it is clear for the programs under evaluation that probability-based interpretation selects more difficult to kill mutants on average than difference-based interpretation.

Table 5.11: Average Killing Frequency Results

Programs	Difference-based metric				Probability-based metric			
	10%	**25%**	**50%**	**100%**	**10%**	**25%**	**50%**	**100%**
FourBalls	24.27%	31.80%	36.32%	31.65%	20.98%	20.73%	22.64%	31.65%
TriTyp	15.66%	19.03%	13.50%	16.77%	14.69%	15.13%	16.16%	16.77%
TCAS	3.68%	3.29%	3.19%	3.13%	1.70%	1.98%	3.08%	3.13%
Mean	**14.5%**	**18.0%**	**17.7%**	**17.2%**	**12.5%**	**12.6%**	**14.0%**	**17.2%**

Figures 5.7 and 5.8 present the mutation score of each selection relative to the complete set of mutants. Probability-based interpretation shows a stronger trend than difference-based interpretation in that selecting fewer (more semantically similar) mutants produces a lower mutation score. Figures 5.10 and 5.11 corroborate this finding with similar results for average killing frequency. Selecting smaller sets of mutants using probability-based interpretation reduces the killing frequency for all three programs, whereas difference-based interpretation actually increases the frequency for certain selection sizes. Probability-based interpretation provides stronger correlations between selection size and mutation score/killing frequency. This indicates that, for the programs under evaluation, it is more effective at determining which mutants are difficult to kill.

Probability-based interpretation achieves success partly due to its superior handling of branch conditions. Rather than assuming paths contribute equally towards a program's semantics, probability-based interpretation weights each path according to how likely it is to be exercised. Most paths through FourBalls have the same probability, whereas triangle types in TriTyp have different probabilities of occurrence. TCAS contains paths that are only exercised when action needs to be taken to avoid a collision. As a result, the difference between Pearson correlation coefficients in Table 5.12 is proportionately higher for TCAS than TriTyp or FourBalls. Probability-based interpretation has six times the correlation coefficient for TCAS (between mutation score and selection size) than difference based-interpretation (see Figure 5.9). Difference-based interpretation has a negative correlation (between killing frequency and selection size) for TCAS, whereas probability-based interpretation has a strong positive correlation (see Figure 5.12). Probability-based interpretation performs better than difference-based interpretation on programs that rely heavily on branch conditions.

Table 5.12: Pearson Correlation Coefficients

Programs	Difference-based metric		Probability-based metric	
	Mutation score	**Frequency**	**Mutation score**	**Frequency**
FourBalls	0.548	0.477	0.850	0.956
TriTyp	0.183	-0.073	0.948	0.961
TCAS	0.130	-0.796	0.780	0.869
Mean	**0.159**	**-0.131**	**0.857**	**0.881**

Summary for RQ8: Probability-based interpretation has been shown to be a more reliable metric of the difficulty involved with killing mutants three Java programs than difference-based interpretation. It can can be used to select mutants, such that random testing achieves a lower mutation score on average than with difference-based interpretation and it is effective for all sizes of selection. Probability-based selection is particularly effective for programs that have complex branch structure (e.g. TCAS). This makes the technique well suited to the task of reducing the number of subdomains that have to be be removed from multiple sets of optimised subdomains.

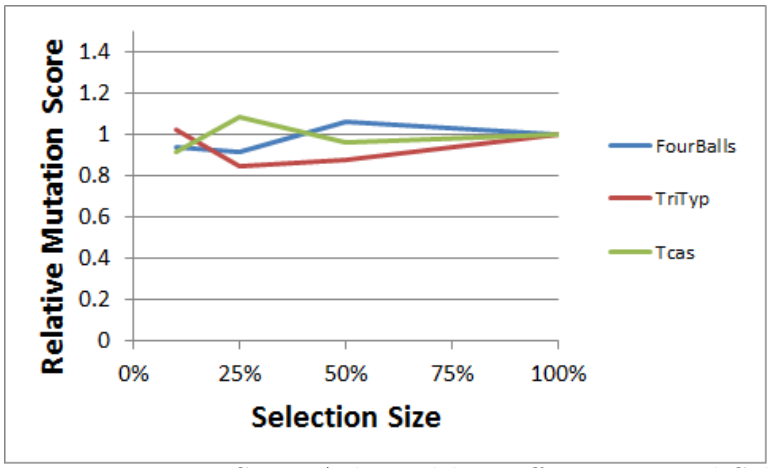


Figure 5.7: Mutation Score Achieved by Difference-Based Selection

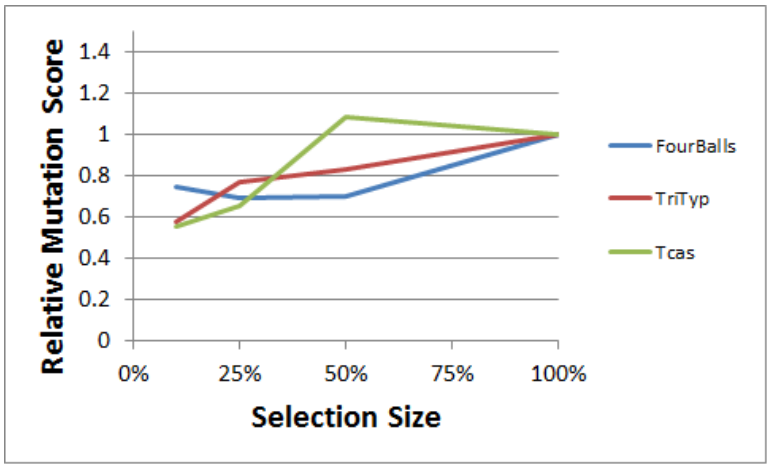


Figure 5.8: Mutation Score Achieved by Probability-Based Selection

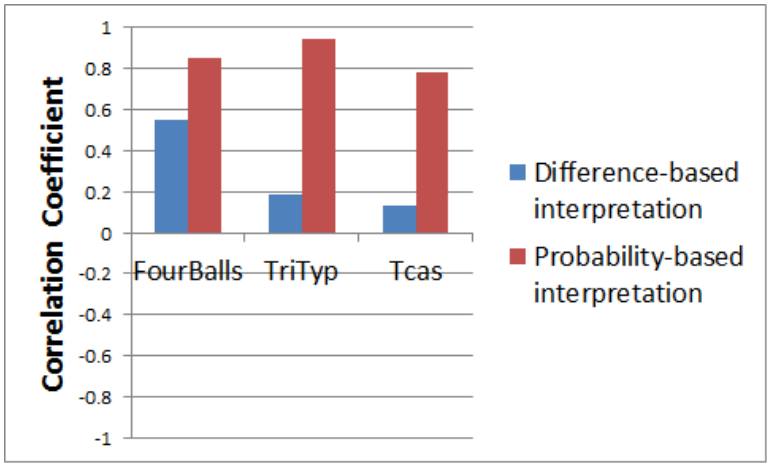


Figure 5.9: Mutation Score Correlations for Each Selection Technique

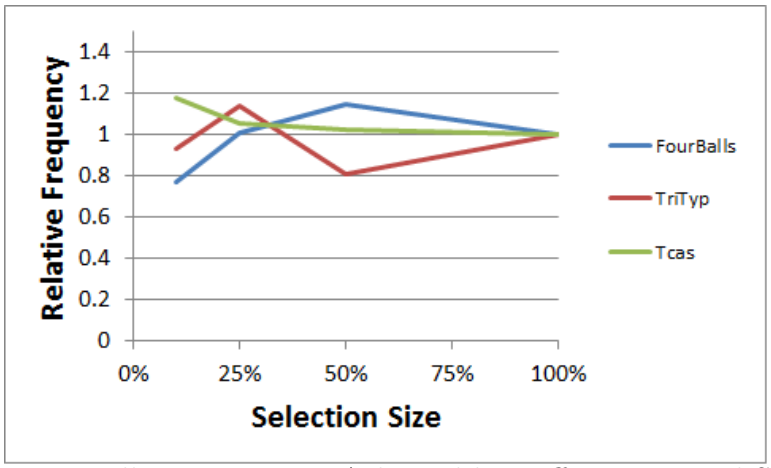


Figure 5.10: Killing Frequency Achieved by Difference-Based Selection

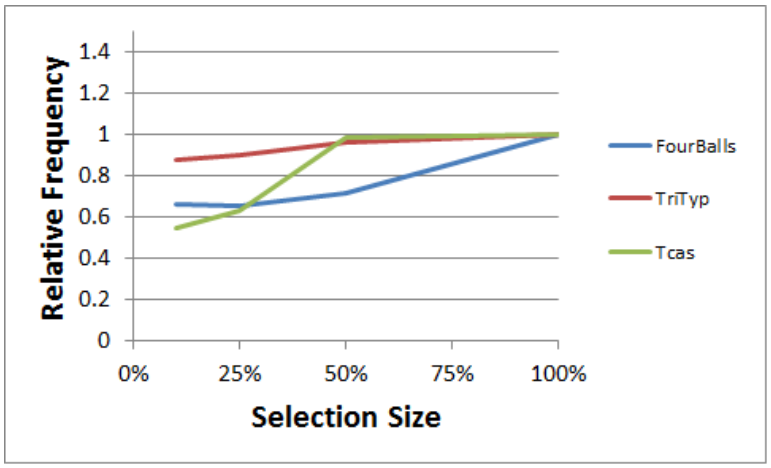


Figure 5.11: Killing Frequency Achieved by Probability-Based Selection

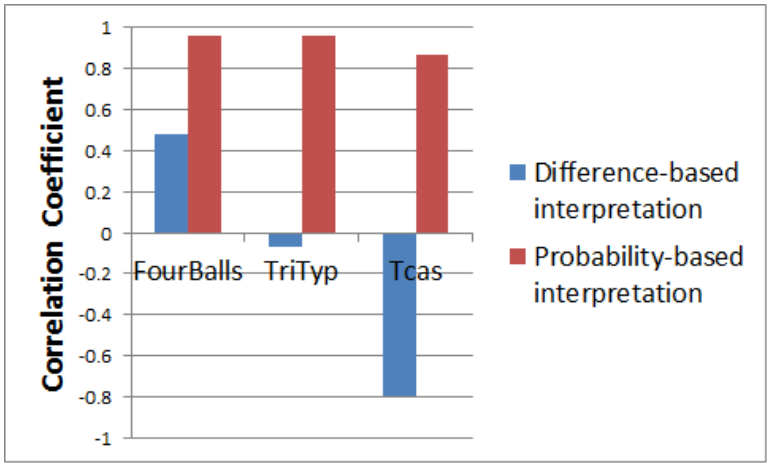


Figure 5.12: Killing Frequency Correlations for Each Selection Technique

5.3.11 Results for RQ9

RQ9 is answered by applying subdomain optimisation to produce multiple sets of optimised subdomains that are effective at killing as many of the mutants selected by probability-based interpretation as possible. Subdomain optimisation is applied on each mutant selection size from RQ8 (10%, 25%, 50% and 100%), selecting mutants in order of how difficult they are to kill. I record the mutation scores achieved by the optimised subdomains, along with the number of subdomains that can be removed once subdomain optimisation is completed. A detailed breakdown of the results can be found in Table 5.13.

Figure 5.13 plots the percentage of subdomains that can be removed after subdomain optimisation with each selection of mutants. The general trend for all three programs is that fewer subdomains can be removed without affecting the mutation score if the subdomains are evolved using fewer mutants. This means probability-based interpretation can be used to reduce the computational expense of subdomain optimisation. The rate at which the percentage of subdomains that can be removed decreases is different for each program (FourBalls decreases the most slowly). Semantic interpretation is more effective for TriTyp and TCAS than FourBalls because they contain more difficult to kill mutants.

Figure 5.14 plots the mutation scores achieved by test suites sampled from within subdomains optimised for each selection of mutants. Optimising subdomains with fewer mutants results in a lower mutation score being achieved (as evaluated on the complete set of mutants). However, even with 10% of the available mutants, the mutation score is still higher than unoptimised random testing. The rate of decrease is non-linear: acceptably high mutation scores may be achieved with 25% or 50% of the mutants. This means that semantic interpretation can be used to reduce the computational expense of subdomain optimisation without overly damaging the fault-finding capability of the resulting test suites.

Summary for RQ9: Subdomains may be evolved to generate effective test suites, but many of the optimised subdomains can be removed without affecting the mutation score (see Chapter 4). Probability-based interpretation provides an efficient alternative to subdomain selection. By optimising subdomains for only the most difficult to kill mutants, a significant reduction in computational expense can be achieved with only a slight decrease in mutation score.

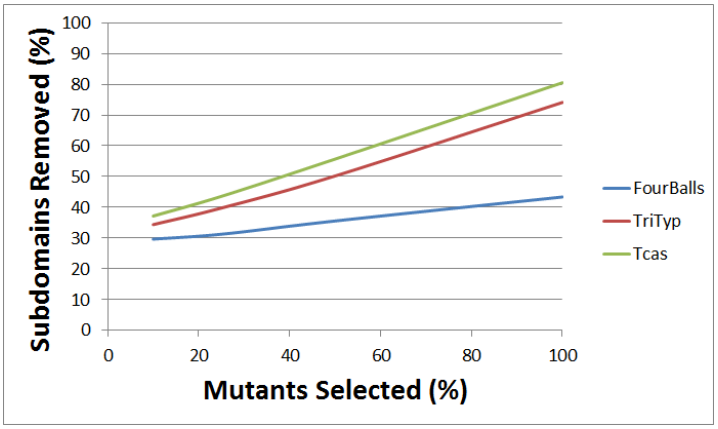


Figure 5.13: Effect of Mutant Selection on Potential for Subdomain Reduction (Averaged over 100 Trials)

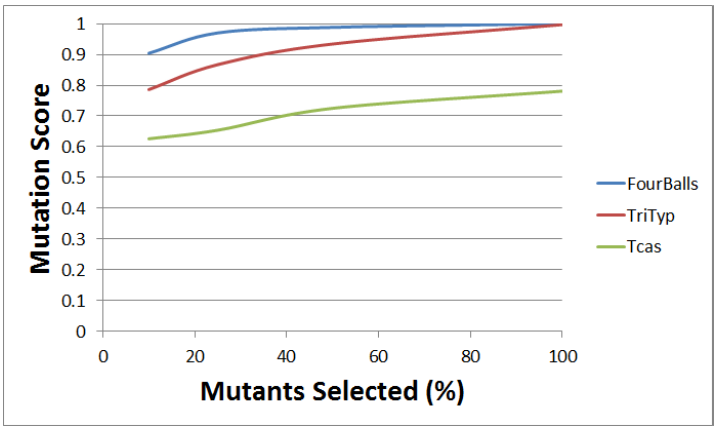


Figure 5.14: Mutation Score of Subdomains on Different Selections of Mutants (Averaged over 100 Trials)

Table 5.13: Effect of Mutant Selection on Subdomain Optimisation

Programs	Mutation Score				Subdomain Reduction			
	10%	**25%**	**50%**	**100%**	**10%**	**25%**	**50%**	**100%**
FourBalls	0.904	0.970	0.988	1.00	0.297	0.313	0.356	0.434
TriTyp	0.786	0.867	0.934	0.997	0.344	0.398	0.502	0.741
TCAS	0.626	0.654	0.725	0.781	0.372	0.436	0.557	0.805
Mean	**0.772**	**0.803**	**0.882**	**0.926**	**0.338**	**0.382**	**0.472**	**0.660**

5.4 Summary

Semantic interpretation uses static analysis to select mutants that are semantically similar to the original program. It does this by executing the program under test symbolically in order to determine the effect that mutations of the program code have on the relationship between inputs and outputs. The competent programmer hypothesis suggests these mutants are particularly useful for testing because they resemble faults that a programmer is more likely to make.

Semantic interpretation may also be used to reduce the computational expense and human effort involved with optimising and selecting efficient sets of subdomains using the techniques described in Chapters 3 and 4. Subdomains that are evolved to kill semantically similar mutants often kill other mutants as a side effect. This means fewer subdomains are needed to achieve a high mutation score and less effort is required to determine the correct output of each test case.

I introduced two semantic interpretation techniques. Difference-based interpretation compares the range of symbolic outputs. It does not take into account path conditions and can only be applied to numerical values, but it is straightforward and computationally inexpensive. Probability-based interpretation evaluates the likelihood the output from a mutant is the same as the original program. It can be used on strings, Booleans, bitwise values and compound objects.

Difference-based interpretation selected difficult to kill mutants effectively for most (but not all) methods evaluated from the Java standard library. Mutants in the top quarter of those selected were less than half as likely to be killed than the average mutant in the remaining three quarters. Probability-based interpretation was shown to select difficult to kill mutants for three more complicated Java programs more effectively than difference-based interpretation. For example, probability-based interpretation achieved 0.78 correlation between selection size and mutation score (compared to 0.13 for difference-based interpretation).

I used probability-based selection to select mutants for subdomain optimisation. When subdomains were optimised for 25% of the (most difficult to kill) mutants, less than 38% of the subdomains had to be discarded (compared to 66% with the complete set). This constitutes a significant reduction in computational expense with a small decrease in mutation score. Semantic interpretation can therefore be used to make subdomain optimisation more efficient.

Chapter 6

Conclusions

6.1 Summary of Achievements

The main aim of this thesis was to make software testing more efficient by introducing new techniques for identifying and evaluating efficient sets of input subdomains for test data generation. This was achieved through a combination of random testing, mutation analysis, metaheuristic optimisation and static analysis. The detailed aims and objectives of this thesis were as follows:

1. Apply subdomain optimisation so as to reduce the number of test cases and input regions that must be evaluated by software testers.

2. Specialise subdomains at killing different groups of mutants, so that they can cover the input domain more effectively and efficiently.

3. Investigate the capability of static program analysis to reduce the number of subdomains that are required to test software effectively.

The first research objective was addressed by optimising input subdomains for 8 Java programs that are frequently used in software testing research. I applied a $1 + 1$ evolution strategy to optimise one subdomain for each input parameter, such that efficient test suites can be sampled randomly from within the bounds of each subdomain. The optimisation technique identified sets of subdomains from which test cases can be selected with higher mutation score than random testing. It only failed to surpass the expected mutation score when it was 100%.

Optimising a single set of subdomains for each program achieved comparable results to dynamic symbolic execution (DSE). Subdomain optimisation achieved a higher mutation score for two programs (TriTyp and Schedule) and a lower mutation score for two others (Tcas and Replace). The programs for which dynamic symbolic execution performed better than subdomain optimisation have complex branch structures. Subdomain optimisation produced less than 1/20th of the test cases used by DSE and one of the weaknesses of optimising a single set of subdomains is that it has to make a trade-off between killing each mutant. Subdomain optimisation therefore struggles to meet the conditions of every branch efficiently.

The subdomains identified by subdomain optimisation serve as a starting point for regression testing, but it can be computationally expensive if the resulting subdomains are only used once. Subdomain optimisation works for the most part as a black-box technique, but it may not always be possible to optimise efficient subdomains without at least some understanding of the program code. For example, none of the initial trials for Tcas produced a high mutation score because of a number of difficult to meet branch conditions involving large constants. Scaling the values of the constants corrected this in part, but it was much more effective to use multiple sets of subdomains.

The second research objective was addressed by optimising multiple sets of subdomains for programs that did not achieve a high mutation score with single sets and two new mathematical Java programs. Sampling test cases from multiple sets of optimised subdomains achieved on average 33% higher mutation score than single sets and 230% higher than unoptimised random testing. One key finding is that multiple sets of subdomains were only slightly more computationally expensive to evolve than single sets. Multiple sets took less time for some programs and more time for others to evolve than single sets.

Multiple sets of subdomains were particularly effective at evolving subdomains for Tcas. As each set of subdomains is able to focus on a different group of mutants, it is not necessary to manually scale the input parameters. I also included three new techniques for automated program stretching. In contrast to manual transformations, program stretching automatically makes mutants easier to kill, then gradually transforms them back again. On Tcas, multiple sets of subdomains and program stretching achieved on average 71% higher mutation score than single sets and took 1/7th of the time to evolve.

The only downside of evolving multiple sets of subdomains is that they can increase the human effort involved with evaluating their test cases. After subdomain optimisation has been completed, small sets of subdomains are selected using sequential floating forward selection (SFFS) so they can kill mutants more efficiently. Subdomain selection reduced the number of subdomains for four out of six programs with little effect on the mutation score. Subdomain selection is computationally inexpensive, but it wastes sets of subdomains that are expensive to evolve. It would be more efficient not to evolve these subdomains at all.

The third research objective was addressed using two static semantic interpretation techniques. Difference-based interpretation compares to output range of mutants with the original program using conservative estimates made by symbolic execution. Probability-based interpretation compares the likelihood the output from a mutant is the same as the original program, assuming probabilities are independent and the output is uniform. Probability-based interpretation achieved a higher correlation between selection size and mutation score (with one million random test cases) than difference-based interpretation ($\rho = 0.78$ compared to $\rho = 0.13$) and can be applied to more data types, but is slightly more expensive.

The competent programmer hypothesis suggests that difficult to kill mutants are more useful because they resemble faults programmers are more likely to make. They can also be used to reduce the computational expense involved with optimising multiple sets of subdomains. Experiments showed that subdomains are more likely to be selected if they are evolved later (i.e. more difficult to kill). When subdomains were optimised for 25% of the (most difficult to kill) mutants, less than 38% of the subdomains had to be discarded (compared to 66% with the complete set). This constitutes a significant reduction in computational expense (as those subdomains no longer had to be optimised) with small a decrease in mutation score. Semantic interpretation can therefore be used to identify difficult to kill mutants and make the process of subdomain optimisation more efficient.

In conclusion, this thesis presents a complete strategy for subdomain-based testing, from the initial static analysis through to optimisation and selection. The evolved subdomains may be sampled from at random to achieve a high mutation score or used by the tester to provide valuable information for regression testing. Compared to other test data generation techniques for mutation analysis (e.g. DSE), subdomain optimisation is less expensive and more effective.

6.2 Limitations of my Research

1. If test cases are only sampled from within the optimised subdomains, it is possible to miss mutants that are difficult to kill because subdomains have not been evolved for them. I address this by evolving multiple sets of subdomains for different groups of mutants. Yet, it may still be necessary to include some test cases sampled from outside the subdomains.

2. Although optimised subdomains are efficient at killing mutants, the process of subdomain optimisation is computationally expensive. Further experiments are required to determine whether it is more efficient at killing mutants (in terms of actual evaluations) than random testing. Yet, the purpose of this research is not simply to kill mutants, but rather to identify small sets of subdomains that are easy to comprehend by the tester.

3. Program stretching makes difficult to kill mutants easier to kill by widening branch conditions for paths along which that mutant was previously killed. This requires the mutant to be killed at least once before it can be useful. Difficult to kill mutants are less likely to be killed during optimisation. It may therefore be helpful to adapt the static analysis techniques in Chapter 5 to guide test data generation along paths more likely to kill the mutant.

4. Probability-based interpretation was shown to be more effective than difference-based interpretation. Yet, it still assumes that the output of a program is uniformly distributed. In practice, program outputs are non-uniform. For example, the output of the numerical expression $a * b$ is sparsely and unevenly generated. It produces highly factorisable numbers more frequently and only generates prime numbers if the value of a or b is one. As a result, probability-based interpretation approximates mutant semantics.

5. The experiments presented in this thesis were performed on a small number of small programs. Further research is required to determine whether these techniques are also effective for larger programs. Work is needed to make the techniques more efficient, so that they are computationally feasible for industrial-scale programs. It is also necessary to consider how subdomains can be optimised for non-procedural (object-oriented) programs.

6.3 Future Work

This section makes some suggestions for future work that will help to address the limitations of my research, as described in the previous section of this thesis.

6.3.1 Optimising Distributions for Entire Input Domain

One of the limitations of my research is that it is necessary to sample outside the evolved subdomains occasionally to test software thoroughly. My current approach does not provide any information about when to do this or which values outside the subdomains are most likely to be useful. One solution to this problem is to optimise input distributions that cover the entire input domain rather than subdomains which just cover certain key areas. Techniques to achieve this may be borrowed from the field of statistical testing (see Section 2.3.6). For example, Poulding et al. [132] use stochastic grammars to describe the input distribution.

Another advantage of this extension to my optimisation technique is that it allows each test case evaluation to have a more direct impact on the shape of the input distribution than is possible by using a CMA-ES to evolve sets of subdomains. Stochastic grammars are more expressive than subdomains and can take into account more of the information gained by test case evaluation. One downside is that more computational resources may be required to optimise stochastic grammars. Another important question is whether a (potentially non-smooth) input distribution is as straightforward for test engineers to understand.

6.3.2 Static Analysis Heuristic for Mutant Propagation

Program stretching currently relies on a mutant previously being killed in order to guide the optimisation process towards subdomain values that are more likely to propagate its effect to the output. One way to address this is to use the static analysis techniques developed for semantic interpretation to predict which paths through a program are more likely to allow a fault to be propagated. The likelihood of fault propagation differs depending on the mutant that needs to be propagated. It would therefore be necessary to apply static analysis whenever a difficult to kill mutant is encountered. The technique must identify paths that are potentially useful, then guide subdomain optimisation along these paths.

Probability-based interpretation can be used to determine for each path the likelihood that the output of a mutant will be different to the original program. It is then necessary to cause subdomain optimisation to follow paths one of the paths that has been identified. This may be achieved using the existing techniques for path stretching and branch-condition stretching. A more advanced strategy would be to automatically infer from the static analysis which subdomain values are most likely to be effective.

6.3.3 Distribution-based Semantic Interpretation

If the output distribution for a program is sparse (as in multiplication) or skewed (as in division), semantic interpretation will underestimate the likelihood that a mutant outputs the same value as the original program. This is because there are less distinct values in the output domain, so it is more likely that two values sampled at random will be the same. This may be addressed by taking into account the shape and sparsity of the output distribution in semantic interpretation.

It is difficult to estimate the proportion of distinct outputs for a given multiplication due to its relationship with prime numbers. If it were possible to identify gaps in the output without calculating every possible value, locating prime numbers would be trivial. As it is, prime numbers are difficult to predict. The best we can do is to use statistical trials and curve fitting to construct an approximate model over a particular range of input for multiplication.

Take for example, subdomains with lower boundary 1 and upper boundary between 1 and 100. I conducted 1000 trials, each with two subdomains randomly sampled from within this range, then calculated every possible multiplication of their values. The proportion of distinct outputs can be modelled using the logarithmic equation: $-0.08\ ln(d) + 0.99$, where d is the total number of outputs produced by the multiplications. This approximation is only valid for subdomains that lie within the range included in the experiment. For subdomains outside this range, it would be necessary to perform more curve fitting experiments. The computational expense involved suggests that this technique may not be worthwhile. Probability-based interpretation may be preferable because, even though it is not entirely accurate, it provides useful information in most cases.

Appendix A

Mutation Operators

A.1 Mutation Operators used by MuJava

Operator	**Description**
AOR	Arithmetic Operator Replacement
AOI	Arithmetic Operator Insertion
AOD	Arithmetic Operator Deletion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
COI	Conditional Operator Insertion
COD	Conditional Operator Deletion
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOI	Logical Operator Insertion
LOD	Logical Operator Deletion
ASR	Assignment Operator Replacement

A.2 Mutation Operators used by Mothra

Operator	Description
AAR	array reference for array reference replacement
ABS	absolute value insertion
ACR	array reference for constant replacement
AOR	arithmetic operator replacement
ASR	array reference for scalar variable replacement
CAR	constant for array reference replacement
CNR	comparable array name replacement
CRP	constant replacement
CSR	constant for scalar variable replacement
DER	DO statement alterations
DSA	DATA statement alterations
GLR	GOTO label replacement
LCR	logical connector replacement
ROR	relational operator replacement
RSR	RETURN statement replacement
SAN	statement analysis
SAR	scalar variable for array reference replacement
SCR	scalar for constant replacement
SDL	statement deletion
SRC	source constant replacement
SVR	scalar variable replacement
UOI	unary operator insertion

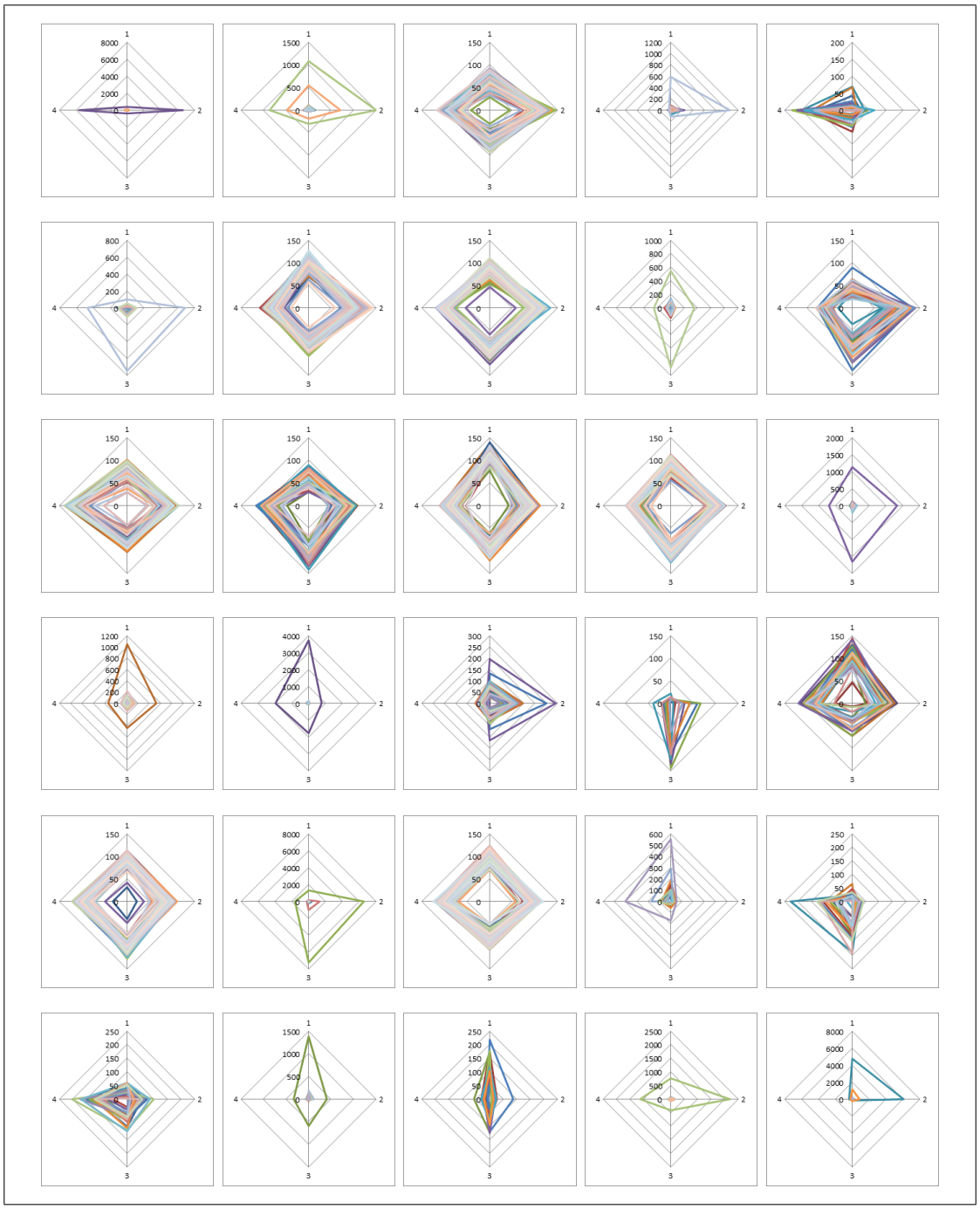
Appendix B

Experimental Data

B.1 Random Seeds (one for each trial)

Single Set of Subdomains	Multiple Sets of Subdomains
995356 957613 815658 501205 249250	751867 126980 375405 358623 795965
94474 347761 348779 361609 851484	25180 442828 997156 322344 573708
849302 277889 214079 152732 851359	975941 618897 901047 451696 249184
902965 33180 57598 412850 179279	755717 979703 594074 387142 559994
242822 57016 92163 94753 312517	720845 767614 209438 116938 171123
850499 630502 804685 45210 484821	302797 505600 423040 261392 961497
141906 157180 664069 808283 799685	115616 931291 587133 598408 199159
626588 528166 663027 918952 988913	673700 880063 53082 710939 424189
941218 737589 265157 420828 696122	864338 964396 871897 85212 401987
822475 248651 596971 575735 367969	334506 677487 849016 423437 258427
155065 189433 543067 467473 886425	990224 151832 472363 504642 389034
471901 778438 128804 253842 740879	100697 425669 391068 355808 460600
612012 472575 862143 808460 530178	377560 638915 659648 275750 28423
942152 36962 903464 650334 790618	551144 175586 711377 909030 115732
257894 583876 36469 114458 345028	735052 268224 931057 143485 909105
370148 923844 883980 583204 825470	340825 605540 157460 339113 396407
928957 57616 543722 371339 350065	264786 894392 638548 318550 336122
886987 602096 216309 783682 145144	931051 447037 638850 130191 446941
277734 713540 796912 481070 812166	674757 864722 49245 534873 95472
443183 317484 58582 534531 733050	172847 974700 63138 287779 567630

B.2 K-means Clustering for SVD Subdomains



(Each figure shows all the sets included in a particular cluster. Each axis represents the normalised subdomain sizes for an input parameter)

Bibliography

[1] A. T. Acree, "On mutation," Ph.D. dissertation, Sch. Inform. Comp. Sci., Georgia Inst. Tech., Atlanta, GA, 1980.

[2] K. Adamopoulos, M. Harman, and R. M. Hierons, "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution," in *Proc. 6th Annu. Conf. Genetic Evolutionary Computation*, 2004, pp. 1338–1349.

[3] K. Akrofi, "Classification of alzheimers disease and mild cognitive impairment by pattern recognition of EEG power and coherence," in *Proc. 35th Int. Conf. Acoustics Speech Signal Processing*, 2010, pp. 606–609.

[4] P. Ammann and J. Offutt, *Introduction to software testing.* New York, NY: Cambridge Univ. Press, 2008.

[5] P. E. Ammann, P. E. Black, and W. Majurski, "Using model checking to generate tests from specifications," in *Proc. 2nd Int. Conf. Formal Eng. Methods*, Brisbane, Australia, 1998, pp. 46—54.

[6] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proc. 27th Int. Conf. Softw. Eng.*, St. Louis, MO, 2005, pp. 402–411.

[7] J. H. Andrews, S. Halder, Y. Lei, and F. C. H. Li, "Tool support for randomized unit testing," in *Proc. 1st Int. Works. Random Testing*, 2006, pp. 36–45.

[8] K. Ayari, S. Bouktif, and G. Antoniol, "Automatic mutation test input data

generation via ant colony," in *Proc. 9th Annu. Conf. Genetic Evolutionary Computation*, London, England, 2007, pp. 1074–1081.

[9] B. Babb, F. Moore, S. Aldridge, and M. R. Peterson, "State-of-the-art lossy compression of martian images via the CMA-ES evolution strategy," in *Proc. Int. Soc. Optics Photonics*, 2012, pp. 22–26.

[10] T. Bäck, *Evolutionary Algorithms in Theory and Practice*. Oxford, England: Oxford Univ. Press: Oxford Univ. Press, 1996.

[11] R. Baker and I. Habli, "An empirical evaluation of mutation testing for improving the test quality of safety-critical software," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 787–805, 2013.

[12] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi, "Toward the determination of sufficient mutant operators for C," *Softw. Testing, Verification, Reliability*, vol. 11, no. 2, pp. 113–136, 2001.

[13] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon, "Automatic test cases optimization using a bacteriological adaptation model: Application to .NET components," in *Proc. Int. Conf. Automated Softw. Eng.*, Edinburgh, United Kingdom, 2002, pp. 253–256.

[14] H.-G. Beyer and H.-P. Schwefel, "Evolution strategies: a comprehensive introduction," *Natural Computing*, vol. 1, no. 1, pp. 3–52, 2002.

[15] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When is nearest neighbor meaningful?" in *Proc. 7th Int. Conf. Database Theory*, Jerusalem, Israel, 1999, pp. 217–235.

[16] B. Bezier, *Software Testing Techniques*, 2nd ed. New York, NY: Van Nostrand Reinhold, 1990.

[17] M. Blair, S. Obenski, and P. Bridickas, "Patriot missile software problem," United States General Accounting Office, Tech. Rep. GAO/IMTEC-92-26, 1992.

[18] H. H. Bock, "On some significance tests in cluster analysis," *J. Classification*, vol. 2, no. 1, pp. 77–108, 1985.

[19] B. Botella, A. Gotlieb, and C. Michel, "Symbolic execution of floating-point computations," *Softw. Testing, Verification, Reliability*, vol. 16, no. 2, pp. 97–121, 2006.

[20] L. Bottaci, "A genetic algorithm fitness function for mutation testing," Toronto, Canada, pp. 3–7, 2001.

[21] G. E. P. Box and M. E. Muller, "A note on the generation of random normal deviates," *Ann. Math. Statist.*, vol. 29, no. 2, pp. 610–611, 1958.

[22] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT - a formal system for testing and debugging programs by symbolic execution," in *Proc. Int. Conf. Reliable Softw.*, Los Angeles, CA, 1975, pp. 234–245.

[23] M. Brendel, "A quick sequential forward floating feature selection algorithm for emotion detection from speech," in *Proc. 11th Annu. Conf. Int. Speech Communication Association*, 2010, pp. 1157–1160.

[24] T. A. Budd, "Mutation analysis of program test data," Ph.D. dissertation, Dept. Comp. Sci., Yale Univ., New Haven, CT, 1980.

[25] C. Cadar and D. Engler, "Execution generated test cases: How to make systems code crash itself," in *Proc. 12th Int. Conf. Model Checking Softw.*, Austin, TX, 2005, pp. 2–23.

[26] F. T. Chan, T. Y. Chen, I. K. Mak, and Y. T. Yu, "Proportional sampling strategy: Guidelines for software testing practitioners," *Inform Softw. Technology*, vol. 38, no. 12, pp. 775–782, 1996.

[27] K. P. Chan, T. Y. Chen, and D. Towey, "Normalized restricted random testing," in *Proc. 8th Int. Conf. Reliable Softw. Technologies*, Toulouse, France, 2003, pp. 368–381.

[28] K. P. Chan, T. Y. Chen, and D. Towey, "Restricted random testing: Adaptive random testing by exclusion," *Int. J. Softw. Eng. Knowledge Eng.*, vol. 16, no. 4, pp. 553–584, 2006.

[29] T. Y. Chen, D. H. Huang, and F.-C. Kuo, "Adaptive random testing by balancing," in *Proc. 2nd Int. Works. Random Testing*, Atlanta, GA, 2007, pp. 2–9.

[30] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and S. P. Ng, "Mirror adaptive random testing," *Inform. Softw. Technology*, vol. 46, no. 15, pp. 1001–1010, 2004.

[31] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," in *Proc. 9th Int. Asian Comp. Sci. Conf.*, Chiang Mai, Thailand, 2005, pp. 320–329.

[32] T. Y. Chen and Y. T. Yu, "On the relationship between partition and random testing," *IEEE Trans. Softw. Eng.*, vol. 20, no. 12, pp. 977–980, 1994.

[33] I. Ciupa, A. Leitner, Oriol, and B. M. Meyer, "ARTOO: adaptive random testing for object-oriented software," in *Proc. 30th Int. Conf. Softw. Eng.*, Leipzig, Germany, 2008.

[34] J. A. Clark, H. Dan, and R. M. Hierons, "Semantic mutation testing," in *Proc. 5th Int. Works. Mutation Analysis*, Paris, France, 2010, pp. 100–109.

[35] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Trans. Software Eng.*, vol. 2, no. 3, pp. 215–222, 1976.

[36] D. M. Cohen and S. R. Dalal, "The combinatorial design approach to automatic test generation," *IEEE Software*, vol. 13, no. 5, pp. 83–88, 1996.

[37] P. Cousot, "Abstract interpretation based formal methods and future challenges," *Lecture Notes in Comp. Sci.*, vol. 2000, pp. 138–156, 2001.

[38] S. B. Dantzig, "Maximization of a linear function of variables subject to linear inequalities," in *Activity Analysis of Production and Allocation*, T. C. Koopman, Ed. New York, NY: Wiley, 1951, pp. 339–347.

[39] M. Daran and P. Thévenod-Fosse, "Software error analysis: a real case study involving real faults and mutations," in *Proc. 10th Int. Symp. Softw. Testing Analysis*, San Diego, CA, 1996, pp. 158–171.

[40] M. Davis, "Hilbert's tenth problem is unsolvable," *Amer. Math. Monthly*, vol. 80, no. 3, pp. 233–269, 1973.

[41] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[42] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Trans. Software Eng.*, vol. 17, no. 9, 1991.

[43] E. Dijkstra, "Notes on structured programming," in *Structured programming*, O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds. London, United Kingdom: Academic Press Ltd., 1972, pp. 1–82.

[44] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE Trans. Softw. Eng.*, vol. 10, no. 4, pp. 438–444, 1984.

[45] E. Dustin, J. Rashka, and J. Paul, *Automated Software Testing: Introduction, Management, and Performance*. Boston, MA: Addison-Wesley Professional, 1999.

[46] M. Ellims, D. Ince, and M. Petre, "The Csaw C mutation tool: Initial results," in *Proc. 3th Int. Works. Mutation Analysis*, Windsor, United Kingdom, 2007, pp. 185–192.

[47] R. Ferguson and B. Korel, "The chaining approach for software test data generation," *ACM Trans. Softw. Eng. Methodology*, vol. 5, no. 1, pp. 63–86, 1996.

[48] A. R. Ford and T. J. Teorey, *Practical Debugging in C++*. Upper Saddle River, NJ: Prentice Hall, 2002.

[49] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses versus mutation testing: an experimental comparison of effectiveness," *J. Syst. Softw.*, vol. 38, no. 3, pp. 235–253, 1997.

[50] P. G. Frankl and E. J. Weyuker, "Provable improvements on branch testing," *IEEE Trans. Softw. Eng.*, vol. 19, no. 10, pp. 962–975, 1993.

[51] G. Fraser and A. Zeller, "Mutation driven generation of unit tests and oracles," in *Proc. 19th Int. Symp. Softw. Testing Analysis*, Trento, Italy, 2010, pp. 147–158.

[52] Fujitsu. (2010) Fujitsu develops technology to enhance comprehensive testing of Java programs. [Online]. Available: http://www.fujitsu.com/global/news/pr/archives/month/2010/20100112-02.html

[53] K. Ghani and J. A. Clark, "Widening the goal posts: Program stretching to aid search based software testing," in *Proc. 1st Int. Symp. Search Based Softw. Eng.*, 2009.

[54] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proc. Conf. Programming Language Design Implementation*, Chicago, IL, 2005, pp. 213–223.

[55] S.-D. Gouraud, A. Denise, M.-C. Gaudel, and B. Marre, "A new way of automating statistical testing methods," in *Int. Conf. Automated Softw. Eng.*, San Diego, CA, 2001, pp. 5–12.

[56] J. S. Gourlay, "A mathematical framework for the investigation of testing," *IEEE Trans. Softw. Eng.*, vol. SE-9, no. 6, pp. 686–709, 1983.

[57] B. J. M. Grün, D. Schuler, and A. Zeller, "The impact of equivalent mutants," in *Proc. 4th Int. Works. Mutation Analysis*, Denver, CO, 2009, pp. 192–199.

[58] M. Haahr. (2013) Random.org: True random number service. [Online]. Available: http://http://www.random.org/

[59] D. Hamlet, "When only random testing will do," in *Proc. 1st Int. Works. Random Testing*, Portland, ME, 2006, pp. 1–9.

[60] D. Hamlet and R. Taylor, "Partition testing does not inspire confidence," *IEEE Trans. Softw. Eng.*, vol. 16, no. 12, pp. 1402–1411, 1990.

[61] R. Hamlet, "Theoretical comparison of testing methods," *ACM SIGSOFT Softw. Eng. Notes*, vol. 14, no. 8, pp. 28–37, 1989.

[62] N. Hansen, "The cma evolution strategy: A comparing review," in *Towards a New Evolutionary Computation (Studies in Fuzziness and Soft Computing)*, J. A. Lozano, P. Larrañaga, I. Inza, and E. Bengoetxea, Eds. Berlin, Germany: Springer, 2006, pp. 75–102.

[63] N. Hansen, A. Auger, R. Ros, S. Finck, and P. Posik, "Comparing results of 31 algorithms from the black-box optimization benchmarking BBOB-2009," in *Proc. 12th Genetic Evolutionary Computation Conf.*, 2010, pp. 1689–1696.

[64] M. Harman, Y. Jia, and W. B. Langdon, "A manifesto for higher order mutation testing," in *Proc. 5th Int. Works. Mutation Analysis*, Paris, France, 2010, pp. 80–89.

[65] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *Proc. 8th Joint Meeting European Softw. Eng. Conf. ACM SIGSOFT Symp. Foundations Softw. Eng.*, Szeged, Hungary, 2011, pp. 212–222.

[66] M. Harman and P. McMinn, "A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation," in *Proc. 16th International Symp. Softw. Testing Analysis*, London, United Kingdom, 2007, pp. 73–83.

[67] I. Harvey, "Artificial evolution: A continuing SAGA," in *Evolutionary Robotics. From Intelligent Robotics to Artificial Life*, T. Gomi, Ed., 2001, pp. 94–109.

[68] L. Hatton, *Safer C: Developing Software for High-Integrity and Safety-Critical Systems*. London, United Kingdom: McGraw-Hill Professional, 1994.

[69] K. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, "A practical tutorial on modified Condition/Decision coverage," NASA, Tech. Rep. NASA/TM-2001-210876, 2001.

[70] W. C. Hetzel, *The Complete Guide to Software Testing*, 2nd ed. Hoboken, NJ: Wiley, 1993.

[71] M. Heusser. (2012, August) Software testing lessons learned from knight capital fiasco. [Online]. Available: http://www.cio.com/article/713628/Software_Testing_Lessons_Learned_From_Knight_Capital_Fiasco/

[72] R. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," *Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 233—262, 1999.

[73] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, 2004.

[74] S. Hussain, "Mutation clustering," Master's thesis, Dept. Comp. Sci., King's College London, London, United Kingdom, 2008.

[75] H. Hutchins, M. andFoster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proc. 16th Int. Conf. on Softw. Eng.*, Sorrento, Italy, 1994, pp. 191–200.

[76] D. C. Ince, "The automatic generation of test data," *Comp. J.*, vol. 30, no. 1, pp. 63–69, 1987.

[77] F. Jay and R. Mayer, "IEEE standard glossary of software engineering terminology," IEEE, Tech. Rep. 610.12-1990, 1990.

[78] Y. Jia and M. Harman. (2010) Mutation testing repository. [Online]. Available: http://crestweb.cs.ucl.ac.uk/resources/mutation_testing_repository/

[79] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, 2011.

[80] J. Jung, "Using evolution strategy for cooperative focused crawling on semantic web," *Neural Comput. Appl.*, vol. 18, no. 3, pp. 213–221, 2009.

[81] D. Kim, "Determination of steel quality based on discriminating textural feature selection," *Chemical Engineering Science*, vol. 66, no. 23, pp. 6264–6271, 2011.

[82] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[83] B. Korel, "Automated software test data generation," *IEEE Trans. Softw. Eng.*, vol. 16, no. 8, pp. 870–879, 1990.

[84] B. Korel, "Dynamic method for software test data generation," *Softw. Testing, Verification, Reliability*, vol. 2, no. 4, pp. 203–213, 1992.

[85] V. Koutavas and M. Wand, "Small bisimulations for reasoning about higher-order imperative programs," in *Proc. 33rd Int. Symp. Principles Programming Languages*, Charleston, SC, 2006, pp. 141–152.

[86] D. R. Kuhn, R. Kacker, and Y. Lei, "Random vs. combinatorial methods for discrete event simulation of a grid computer network," in *Proc. Modelling Simulation World Conf. Expo.*, Virginia Beach, VA, 2009.

[87] D. R. Kuhn and M. Reilly, "An investigation of the applicability of design of experiments to software testing," in *Proc. 27th Annu. Softw. Eng. Works.*, Greenbelt, MD, 2002, pp. 91–95.

[88] R. Kuhn and Y. Lei, "Practical combinatorial testing: Beyond pairwise," *IEEE IT Professional*, vol. 10, no. 3, 2008.

[89] F.-C. Kuo, "On adaptive random testing," Ph.D. dissertation, Fclty. Inform. Commun. Technologies, Swinburne Univ. Technology, Hawthorn, Australia, 2006.

[90] F.-C. Kuo, T. Y. Chen, H. Liu, and W. K. Chan, "Enhancing adaptive random testing in high dimensional input domains," in *Proc. 22nd Annu. Symp. Applied Computing*, Seoul, Korea, 2007, pp. 1467–1472.

[91] K. Lakhotia, "Search based testing," Ph.D. dissertation, Dept. Comp. Sci, King's College London, London, United Kingdom, 2009.

[92] E. Larson and T. Austin, "High coverage detection of input-related security faults," in *Proc. 12th Conf. USENIX Security*, Washington, DC, 2003, pp. 9–24.

[93] N. Li, U. Praphamontripong, and J. Offutt, "An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage," in *4th Int. Works. Mutation Analysis*, Denver, CO, 2009, pp. 220–229.

[94] R. Lipton, "Fault diagnosis of computer programs," Sch. Comp. Sci., Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep., 1971.

[95] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava : an automated class mutation system," *J. Softw. Test. Verif. Rel.*, vol. 15, no. 2, pp. 97–133, 2005.

[96] S. Mahmood, "A systematic review of automated test data generation techniques," Master's thesis, Sch. Eng., Blekinge Inst. Tech., Ronneby, Sweden, 2007.

[97] S. Mankefors, R. Torkar, and A. Boklund, "New quality estimations in random testing," in *14th Int. Symp Softw. Reliability Eng.*, Denver, CO, 2003, pp. 468–478.

[98] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Modeling Comput. Simulation*, vol. 8, no. 1, pp. 3–30, 1998.

[99] P. S. May, "Test data generation: two evolutionary approaches to mutation testing," Ph.D. dissertation, Dept. Comp. Sci., Univ. Kent, Canterbury, United Kingdom, 2007.

[100] J. Mayer, "Adaptive random testing by bisection with restriction," in *Proc. 7th Int. Conf. Formal Eng. Methods*, Manchester, United Kingdom, 2005, pp. 251–263.

[101] J. Mayer, T. Y. Chen, and D. H. Huang, "Adaptive random testing through iterative partitioning revisited," in *Proc. 3rd Int. Works. Softw. Quality Assurance*, Portland, OR, 2006, pp. 22–29.

[102] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, 1976.

[103] P. McMinn, "Search-based software test data generation: A survey," *Softw. Testing, Verification, Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[104] P. McMinn, M. Harman, D. Binkley, and P. Tonella, "The species per path approach to Search-Based test data generation," in *Proc. 15th Int. Symp. Softw. Testing Analysis*, Portland, ME, 2006, pp. 13–24.

[105] P. McMinn, M. Stevenson, and M. Harman, "Reducing qualitative human oracle costs associated with automatically generated test data," in *Proc. 1st Int. Works. Software Test Output Validation*, Trento, Italy, 2010, pp. 1–4.

[106] N. Megiddo, "Linear programming," in *Encyclopedia of Microcomputers*, A. Kent and J. Williams, Eds. Academic Press Ltd., 1991.

[107] C. C. Michael, G. McGraw, and M. A. Schatz, "Generating software test data by evolution," *IEEE Trans. Software Eng.*, vol. 27, no. 12, pp. 1085–1110, 2001.

[108] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Efficient JavaScript mutation testing," in *Proc. 6th Int. Conf. Softw. Testing Verification*, 2013, pp. 74–83.

[109] E. S. Mresa and L. Bottaci, "Efficiency of mutation operators and selective mutation strategies: An empirical study," *Softw. Testing, Verification Reliability*, vol. 9, no. 4, 1999.

[110] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*. Hoboken, NJ: Wiley, 2012.

[111] V. Naidu, "Manual testing versus automated testing," in *Testing Controversies*, A. Sharma, Ed. Bangalore, India: Infosys, 2011, pp. 6–9.

[112] A. S. Namin, J. H. Andrews, and D. J. Murdoch, "Sufficient mutation operators for measuring test effectiveness," in *Proc. 30th Int. Conf. Softw. Eng.*, Leipzig, Germany, 2008, pp. 351–360.

[113] V. Nissen and S. Gold, "Survivable network design with an evolution strategy," in *Success in Evolutionary Computation (Studies in Computational*

Intelligence), A. Yang, Y. Shan, and L. T. Bui, Eds. Berlin, Germany: Springer, 2008, pp. 263–283.

[114] S. Ntafos, "On random and partition testing," in *Proc. 11th Int Symp. Softw. Testing Analysis*, Clearwater Beach, FL, pp. 42–48.

[115] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Trans. Softw. Eng. Methodology*, vol. 1, no. 1, pp. 5–20, 1992.

[116] A. J. Offutt and W. M. Craft, "Using compiler optimization techniques to detect equivalent mutants," *Journal Softw. Testing, Verification, Reliability*, vol. 4, no. 3, pp. 131–154, 1994.

[117] A. J. Offutt and J. H. Hayes, "A semantic model of program faults," in *Proc. 10th Int Symp. Softw. Testing Analysis*, San Diego, CA, 1996, pp. 195—200.

[118] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Trans. Softw. Eng. Methodology*, vol. 5, no. 2, pp. 99–118, 1996.

[119] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Softw. Testing, Verification, Reliability*, vol. 7, pp. 165—192, 1997.

[120] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *Proc. 15th Int. Conf. Softw. Eng.*, Baltimore, MD, 1993, pp. 100–107.

[121] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the orthogonal," in *Mutation Testing for the New Century*, W. E. Wong, Ed. Norwell, MA: Kluwer Academic Publishers, 2001, pp. 34–44.

[122] A. J. Offutt and J. M. Voas, "Subsumption of condition coverage techniques by mutation testing," Dept. Inf. Software Syst. Eng., George Mason University, Tech. Rep., 1996.

[123] Oracle. (2011) Java™ platform, standard edition 7 api specification. [Online]. Available: http://docs.oracle.com/javase/7/docs/api/

[124] Oracle. (2013) Java bug database. [Online]. Available: http://bugs.sun.com

[125] M. Oriol and S. Tassis, "Testing .NET code with YETI," in *Proc. 15th Int. Conf. Eng. Complex Comput. Syst.*, 2010, pp. 264–265.

[126] C. Pacheco, S. Lahiri, and T. Ball, "Finding errors in .NET with feedback-directed random testing," Microsoft Research, Tech. Rep. MSR-TR-2008-29, 2008.

[127] M. R. Paige, "Program graphs, an algebra, and their implication for programming," *IEEE Trans. Softw. Eng.*, vol. SE-1, no. 3, pp. 286–291, 1975.

[128] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in *Proc. 19th Int. Symp. Softw. Testing Analysis*, Trento, Italy, 2010, pp. 121–130.

[129] M. Papadakis and N. Malevris, "Searching and generating test inputs for mutation testing," *SpringerPlus*, vol. 2, no. 121, 2013.

[130] R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test-data generation using genetic algorithms," *Softw. Testing Verification, Reliability*, vol. 9, no. 4, pp. 263–282, 1999.

[131] S. Poulding and J. A. Clark, "Efficient software verification: statistical testing using automated search," *IEEE Trans. Software Eng.*, vol. 36, no. 6, pp. 763–777, 2010.

[132] S. Poulding, J. A. Clark, R. Alexander, and M. Hadley, "The optimisation of stochastic grammars to enable cost-effective probabilistic structural testing," in *Proc. 15th Annu. Conf. Genetic Evolutionary Computation*, 2013, pp. 1477–1484.

[133] C. S. Păsăreanu and N. Rungta, "Symbolic pathfinder: symbolic execution of Java bytecode," in *Proc. 25th Int. Conf. Automated Softw. Eng.*, Antwerp, Belgium, 2010, pp. 179–180.

[134] P. Pudil, F. J. Ferri, J. Novovicova, and J. Kittler, "Floating search methods for feature selection with nonmonotonic criterion functions," in *Proc. 12th Int. Conf. Pattern Recognition*, 1994, pp. 279–283.

[135] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen, "On the automated generation of program test data," in *Proc. 2nd Int. Conf. Softw. Eng.*, San Francisco, CA, 1976, p. 636.

[136] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. Softw. Eng.*, vol. 11, no. 4, pp. 367–375, 1985.

[137] G. Redelinghuys, "Symbolic string execution," Master's thesis, Dept. Comp. Sci., Stellenbosch Univ., Matieland, South Africa, 2012.

[138] S. C. Reid, "An empirical analysis of equivalence partitioning, boundary value analysis and random testing," in *Proc. 4th Int. Symp. Softw. Metrics*, Albuquerque, NM, 1997, pp. 64–73.

[139] D. Schmidt and B. Steffen, "Program analysis *as* model checking of abstract interpretations," *Lecture Notes in Comp. Sci.*, vol. 1503, pp. 351–380, 1998.

[140] D. Schuler and A. Zeller, "(Un-)Covering equivalent mutants," in *Proc. Int. Conf. Softw. Testing, Verification, Validation*, Paris, France, 2010, pp. 45–54.

[141] H.-P. Schwefel, *Evolution and optimum seeking.* New York, NY: Wiley, 1995.

[142] R. M. Stallman, *Using the GNU Compiler Collection.* Boston, MA: GNU Press, 2008.

[143] T. Strutz, *Data Fitting and Uncertainty.* Wiesbaden: Springer, 2010.

[144] Sun Microsystems. (2008, August) Java compatability kit 6b user's guide.

[145] P. Thévenod-Fosse and H. Waeselynck, "STATEMATE applied to statistical software testing," in *Proc. 8th Int. Symp. Softw. Testing Analysis*, Cambridge, MA, 1993, pp. 99–109.

[146] N. Tillmann and W. Schulte, "Unit tests reloaded: Parameterized unit testing with symbolic execution," *IEEE Software*, vol. 23, no. 4, pp. 38–47, 2006.

[147] M. P. Usaola, P. R. Mateo, and B. P. Lamancha, "Reduction of test suites using mutation," in *Proc. 15th Int. Conf. Fundamental Approaches Softw. Eng.*, 2012, pp. 425–438.

[148] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *J. Aut. Softw. Eng.*, vol. 10, no. 2, pp. 3–12, 2003.

[149] J. M. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs Against Errors*. New York, NY: Wiley, 1998.

[150] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Inform. Softw. Technology*, vol. 43, no. 14, pp. 841–854, 2001.

[151] T. Weise, *Global Optimization Algorithms - Theory and Application*, 2009. [Online]. Available: http://www.itweise.de/projects/book.pdf

[152] E. Weyuker, "On testing non-testable programs," *Comput. J.*, vol. 25, no. 4, pp. 465–470, 1982.

[153] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Trans. Softw. Eng.*, vol. 17, no. 7, pp. 703–711, 1991.

[154] L. J. White and E. I. Cohen, "A domain strategy for computer program testing," *IEEE Trans. Softw. Eng.*, vol. SE-6, no. 3, pp. 247–257, 1980.

[155] J. A. Whittaker and M. G. Thomason, "A markov chain model for statistical software testing," *IEEE Trans. Softw. Eng.*, vol. 20, no. 10, 1994.

[156] W. E. Wong and A. P. Mathur, "Reducing the cost of mutation testing: an empirical study," *J. Syst. Software*, vol. 31, no. 3, pp. 185–196, 1995.

[157] M. Ying, "Bisimulation indexes and their applications," *Theoretical Comp. Sci.*, vol. 275, no. 1-2, pp. 1–68, 2002.

[158] S. Yoo and M. Harman, "Regression testing minimisation, selection and prioritisation: a survey," *Softw, Testing, Verification, Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[159] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei, "Test generation via dynamic symbolic execution for mutation testing," in *Proc. 26th Int. Conf. Softw. Maintenance*, Timişoara, Romania, 2010, pp. 1–10.

[160] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Survey*, vol. 29, no. 4, pp. 366–427, 1997.