An Empirical Study of the Regression Testing of an Industrial Software Product

by

Daniel R. D. Di Nardo

A thesis submitted to the

Faculty of Graduate Studies and Research in partial fulfilment

of the requirements for the degree of

Master of Applied Science in Electrical Engineering

Ottawa-Carleton Institute of Electrical and Computer Engineering

Department of Systems and Computer Engineering, Carleton University

Ottawa, Ontario, Canada

September 2007

Your file Votre référence
ISBN: 978-0-494-33643-4
Our file Notre référence
ISBN: 978-0-494-33643-4

ABSTRACT

This study explores four important regression testing methodologies: retest-all, regression test selection, test suite reduction, and test case prioritization. Experiments that run specific techniques to implement each of the test methodologies are conducted; an industrial software package with multiple versions and having real fault data is used. For each technique, the use of fine-grained and coarse-grained code coverage data is considered. For regression test selection: the use of a modification-based selection technique yields no cost savings; the use a modification-focused minimization technique yields savings in test executions times but with a reduction in fault detection effectiveness. For test suite reduction, the use of a coverage-focused minimization technique yields savings in test executions times but with a reduction in fault detection effectiveness. Prioritization techniques using an additional-coverage strategy and fine-grained coverage data improve the fault detection rate of a test set. Only the prioritization techniques are shown to be beneficial.

ACKNOWLEDGEMENTS

I would like to thank Dr. Lionel Briand and Dr. Yvan Labiche for their guidance, editing assistance, and editorial advice. Without their support this thesis would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

APFD	Average of the Percentage of Faults Detected
CCFG	Class Control Flow Graph
FEP	Fault Exposing Potential
FI	Fault Index
ICFG	Interprocedural Control Flow Graph
KLOC	Thousands of Lines of Code
LOC	Lines of Code
OO	Object-Oriented
RTS	Regression Test Selection

NOMENCLATURE

Mathur in [39] provides an overview of the terminology related to software testing.

A *test set* or *test suite* contains zero or more test cases.

A *test case* consists of test data to be inputted to a program and the expected output.

The *test data* is a set of values, one for each input variable.

An *error* might be committed by a programmer writing a program.

A *fault* results from one or more errors.

A *failure* results when test data executes a fault.

1 INTRODUCTION

Modern software development consists primarily of modifying existing software; consequently, software maintenance can account for up to two-thirds of the cost of software production [23].

The goal of regression testing is to revalidate modified software over subsequent releases. Regression testing ensures that new and modified code functions correctly and it also assures that unchanged code continues to function correctly. Software testing is an expensive process and techniques that reduce costs are highly desirable.

A variety of regression testing techniques have been devised in an attempt to reduce the cost of testing modified software versions. Among the proposed cost reducing techniques are approaches that select a subset of the previously released software's test suite or prioritization approaches that reorder the test suite used in the development process.

To date, we do not know the true effectiveness of many regression testing techniques. Many empirical studies have been performed using small programs. Many of the existing studies make use of artificial code changes (i.e., experimenters seed faults in the code) to create "new versions" against which to evaluate the effectiveness of the regression test techniques under study. More empirical studies of large industrial sized programs are desirable to increase the data available to researchers. Studies making use of real world faults would also provide much needed realism compared to the approach of using artificially seeded faults.

1.1 Thesis Contribution

This project is a case study that seeks to evaluate and compare the cost and effectiveness of different regression testing techniques. This is one of the first real case studies to apply several regression testing methodologies to an available industry software package. Several real versions of the software are used and real faults are used to evaluate the effectiveness of the selected regression test techniques.

The following four important regression test methodologies are explored: retest-all, regression test selection, test suite reduction, and test case prioritization. This thesis summarizes the results of several experiments that run specific techniques to implement each of the regression test methodologies. For each of the techniques considered, the use of fine-grained and coarse-grained code coverage data is evaluated.

For regression test selection: the use of a modification-based selection technique yields no cost savings; the use a modification-focused minimization technique yields significant savings in test executions times but with a significant reduction in fault detection effectiveness. For test suite reduction the use of a coverage-focused minimization technique yields significant savings in test executions times but with a significant reduction in fault detection effectiveness. Prioritization techniques using an additional-coverage strategy and fine-grained coverage data were shown to improve the fault detection rate of a test set. Only the prioritization techniques used in this study were shown to be beneficial to the regression testing process.

1.2 Thesis Limitations

This study only makes use of a subset of the existing regression testing techniques and therefore it only constitutes a partial study of the regression testing problem. Those techniques are however among the most commonly used and investigated.

The subject software of this study runs on the Windows platform. The majority of the available regression testing tools are not implemented for the Windows platform. This highly limited the choice of regression testing software tools. The Telcordia *x*Suds software tool that runs on Windows was chosen for the experimental analyses.

Although experimenting on a single software package enables us to generate interesting data for discussion, any claims that can be made pertaining to the resulting data is limited.

1.3 Thesis Organization

This document is organized in the following sections: a discussion of the background and related work in the study of regression test techniques (Section 2), a description of the experiments performed in this study (Section 3), the results and analyses of the experiments (Section 4), and the conclusions drawn from the study (Section 5).

2 BACKGROUND AND RELATED WORK

2.1 Regression Testing Overview

An initial test set *T* is developed for a given program *P*. Regression testing seeks to validate the modified version of *P*, designated as *P'*. Typically, the practice is to reuse *T* to test *P'*. New test cases may of course be required to test the new or changed functionality of *P'*, but in the study of regression testing techniques, it is the reuse of the existing test cases from *T* that we are interested in.

Rothermel et al. in [46] have documented four important regression test methodologies that involve the reuse of existing test cases, namely: retest-all, regression test selection, test suite reduction, and test case prioritization. The new test set resulting from the application of one of the methodologies is designated as *T'*.

Regression testing should ideally be performed at all stages of programming; specifically, tests should be run at the unit level, integration level, and system level when a program is modified. The regression test methodologies can be applied at each of these levels of software development.

2.2 Regression Testing Process

Mathur in [39] summarizes a common subset of tasks involved in regression testing. The following list defines each task in a likely ordering:

1. Test revalidation/selection/minimization/prioritization. Test revalidation is the process of checking which test cases in *P* remain valid for *P'*. Any of the regression testing techniques can be applied following revalidation: selection, minimization, or prioritization.

2. Test set up. Refers to the process by which the system under test is configured such that its environment is ready to receive data and then report the desired output information.

3. Test sequencing. In some systems, the ordering of tests requires attention. An internal state might be required for a given test to function properly. Thus, some initial test sequence might be required to attain the desired state.

4. Test execution. Once the preceding tasks have been performed, the tests need to be executed. An automated tool is necessary to execute tests and record test results reliably, especially given a large application with a large regression test suite (manual execution would not be feasible in this case).

5. Output comparison. The outcome of each executed test needs to be verified. The entity that checks for the correctness of the test outcome is known as an oracle. Typically an oracle will compare the outcome of a test to its desired outcome to determine whether the test has succeeded or failed.

6. Fault mitigation.

This process assumes that P has already been modified and that P' is available for regression testing.

The following figures (Figure 2-1, Figure 2-2, and Figure 2-3) illustrate a possible implementation of the regression testing process. This process is followed for the experiments performed for this thesis (see section 3).

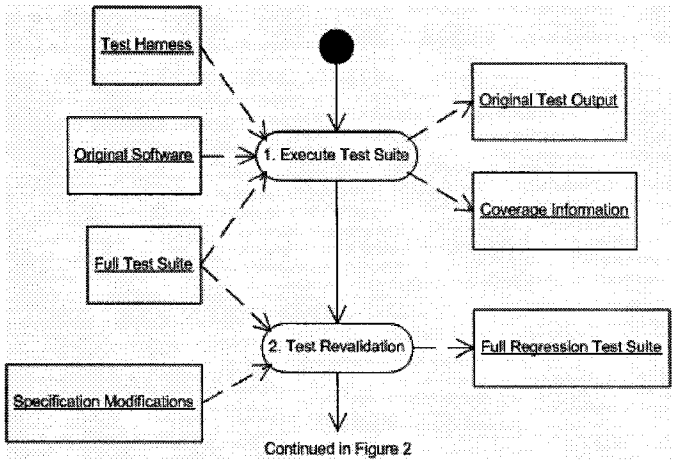


Figure 2-1 – Regression testing process, Part I.
The Full Test Suite is run against the Original Software. The Full Regression Test Suite for the modified software is then determined by revalidating the Full Test Suite of the original software against the specification modifications for the new software release.

Step 1 of Figure 2-1 is actually a precursor step in the regression testing process. It demonstrates what is needed before the application of regression testing techniques can be performed. A test suite (in this experiment a series of test scripts) is initially written for a base version of the software and executed with the aid of a test harness. The test output is examined for its correctness (manually for this experiment) and then saved. For this analysis phase, the software being tested is also instrumented to provide code coverage information; hence, the code coverage information for each test case is recorded.

Thus for each test case in the Full Test Suite there is: a test script that inputs the test data, the expected test output values (in this experiment a corresponding output file is saved for each input test script), and the code coverage information.

Step 2 of Figure 2-1 demonstrates how the Full Regression Test Suite for the next version is determined. Each existing test case must be revalidated before it can be used to test the next version of the software. The software modifications are examined (e.g., from change request specifications). Any test cases that have been rendered obsolete are not included in the full regression test set.

Figure 2-2 demonstrates how one can apply a variety of regression testing techniques to the Full Regression Test Suite. All of the techniques make use of the test case code

Coverage Information for the original software version. For some of the techniques, only the coverage of the modified code is considered; an additional analysis phase (step 6) is required here to get the code Modification Information. Some regression testing techniques using only Coverage Information are: coverage-focused minimization (step 3), total-coverage prioritization (step 4), and additional-coverage prioritization (step 5). Some regression testing techniques using Coverage Information and Modification Information are: modification-based regression test selection (RTS) (step 7), modification-focused minimization (step 8), total-coverage-using-modification-information prioritization (step 9), and additional-coverage-using-modification-information prioritization (step 10). Each of these techniques is described in detail in section 2.4.

A subset (for the test selection or minimizations) or a reordered sequence (for the prioritizations) of the Full Regression Test Suite results after the application of any of these techniques. The regression test suite that results from the application of the regression testing techniques is called the Resulting Regression Test Suite.

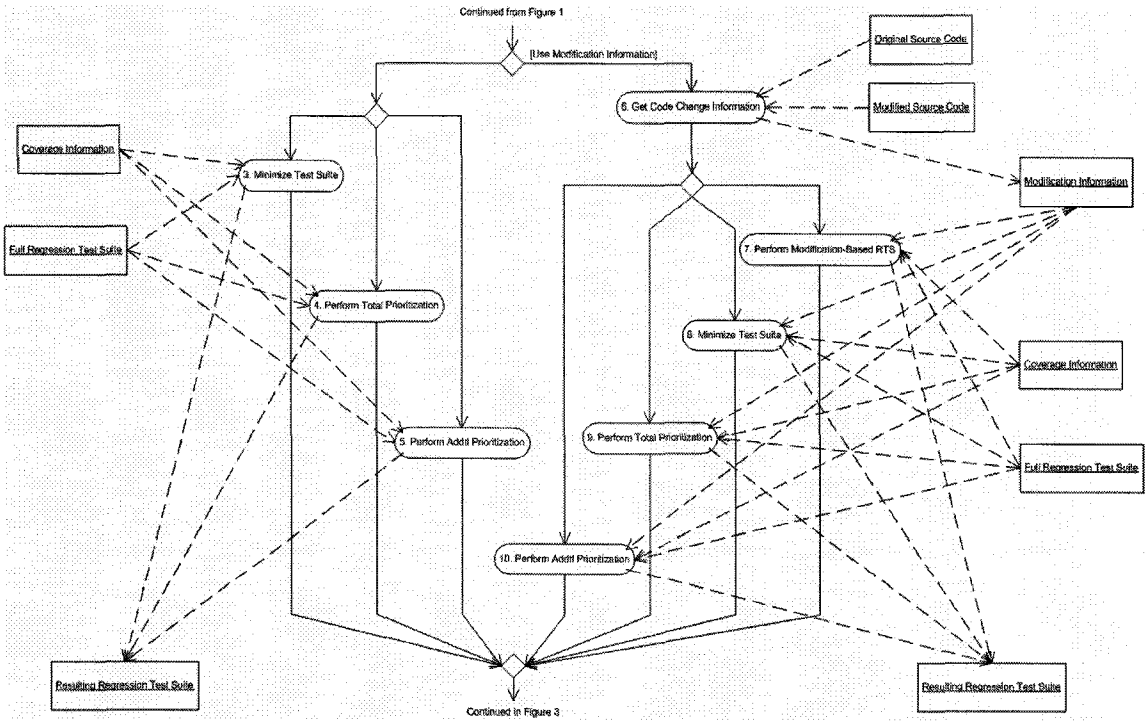


Figure 2-2 – Regression testing process, Part II.
A regression test technique is applied to select a subset of the Full Regression Test Suite.

Step 11 of Figure 2-3 finally shows how the Resulting Regression Test Suite is evaluated against the new software version. The test harness is again used to execute the test cases and the resulting output files are saved. A test oracle compares the regression test output files against the expected output files (saved when testing the original version, Step 1 of Figure 2-1) for each test case. If the files are identical, the test case passes. If the files are different, the test case fails, thus a fault or faults have been detected in the modified software.

Unlike the execution of the original software version (step 1 of Figure 2-1), there is no need to instrument the software to provide any coverage data at this stage; this information is not needed for the regression testing step. Code instrumentation increases execution times and could significantly delay testing during the regression testing phase. Eventually, as explained below, we do collect the new coverage information associated with the test cases, but this can be done after the release of the new software.

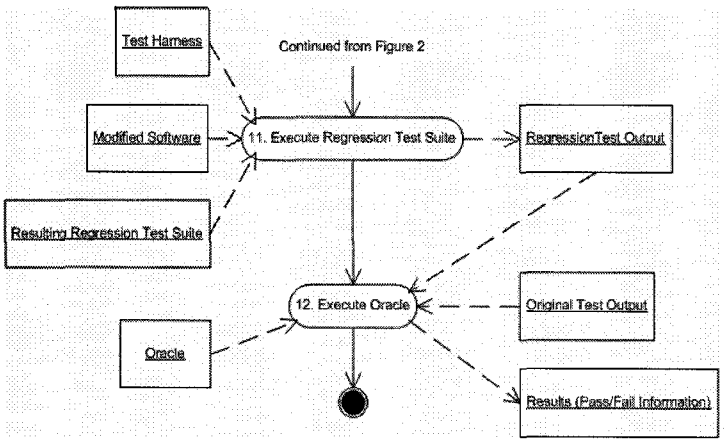


Figure 2-3 – Regression testing process, Part III.
The Resulting Regression Test Suite from the previous step is applied to the modified software. The test output is evaluated.

Once the modified software version has been successfully regression tested and released, we assume that before the next modified version is developed and requires regression testing we have time to repeat the initial analysis of Figure 2-1 on the new Full Test Suite (i.e., the Full Regression Test Suite and any new test cases that were added to test the new or modified requirements). This is necessary to collect new baseline information. Obviously for the new test cases we must save the test output values for future

comparisons with subsequent software versions. Code coverage information is saved for all the test cases of the new Full Test Suite, not just the newly added ones. This is necessary since major code changes can occur between versions and the code coverage data associated with a given test case could have potentially undergone dramatic changes.

2.3 Testing Procedural Programs vs. Testing Object-Oriented Programs

The requirement that modified OO programs be retested is the same as the requirement that modified procedural programs be retested. Chen and Kao [6] demonstrate the need for a different approach when testing object-oriented programs as opposed to procedural programs. Traditional procedural program testing techniques can fail to detect faults that are caused by either inheritance or polymorphism features in OO programming. By extension, regression testing strategies have been created to address not only programs written in procedural languages, but also OO languages.

Rothermel et al. in [52] conduct a study that specifically addresses the testing of OO software. They assert that while many researchers have addressed the regression test selection problem for procedural software, far fewer have addressed the problem for object-oriented software. The emphasis on code reuse present in the object-oriented paradigm actually increases the need for costly regression testing, thus the savings derived from using regression test selection techniques could potentially be substantial. Ideally, when a class is modified every program that uses the class, the modified class itself, and every class derived from the class should be retested.

As stated above, in OO programming, modifications to classes require that additional class level retesting be performed. Class testing techniques typically involve the use of a test driver to invoke sequences of methods in varying orders, and after each sequence, verify that the resulting state of the objects manipulated by the methods is correct. Because most classes function as sets of interacting methods, regression test selection techniques need to focus specifically on testing at the interprocedural level.

2.4 Regression Testing Techniques

As mentioned previously, there are four important regression test methodologies involving the reuse of existing test cases: retest-all, regression test selection, test suite reduction, and test case prioritization. The following subsections provided a detailed description of each of these methodologies; available techniques are presented from the literature.

For reasons explained in section 3.6, the Telcordia Software Visualization and Analysis Toolsuite (*x*Suds) is selected for use in the empirical study performed as part of this thesis. Hence, a summary of *x*Suds supported regression testing techniques used in this study is also presented below. An overview of the *x*Suds tool is presented in section 2.10.2. *x*Suds can be used with C and C++ programs on the UNIX or Windows platforms.

2.4.1 Retest-all

This is a popular technique used in the software industry whereby all existing test cases are rerun [42]. Any obsolete test cases in T are discarded or rewritten when testing P'; obsolete test cases result when test cases in T no long apply to P' [36]. Due to the fact that this technique tends to consume a lot of time and resources [49], the other approaches described herein may provide a desirable alternative to this technique.

2.4.2 Regression test selection

Regression test selection uses information about P, P', and T to select a subset of T to use in order to test P' [46].

Rothermel and Harrold present a summary of code-based regression test selection techniques in [49]. Coverage-based regression test selection techniques seek to identify modified program components and select the tests in T that exercise these components. Minimization techniques work as do coverage-based selection techniques but they additionally focus on minimizing the number of tests run on the modified program components.

Rothermel and Harrold [49] have studied a wide variety of regression test selection techniques, both unsafe and safe; though few of these techniques have actually been implemented. (Section 2.4.2.1 explains the difference between safe and unsafe techniques.) They define a framework for comparing the different methods, based on the following: inclusiveness, the ability to choose modification revealing tests; precision, the ability to eliminate or exclude tests that will not reveal behavioral differences; efficiency, the space and time requirements of the method; and generality, the applicability of the method to different classes of languages and modifications.

Mathur in [39] points out that when attempting to select a subset of a test set, care must be taken to assure that any test interdependencies be considered (e.g., a test case might require that another test case be run first for proper execution). Regression test selection techniques might not be practical should test case dependencies be so complex that a test selection tool cannot incorporate this information.

2.4.2.1 Safe vs. unsafe

There are two distinct subcategories of regression test selection techniques, namely, safe and unsafe. Safe techniques select every test in T that can reveal one or more faults in P' – that is the technique selects every test in T that is modification-traversing for modifications from P to P'; unsafe techniques may discard several tests from T that are modification-traversing – these techniques can often rival the efficacy of the safe selection techniques while dramatically reducing the number of tests that need to be executed [23]. If high reliability is desired we may choose a safe technique even though this might be a more time consuming and costly choice; if there is little time to test we may use a minimization technique even though this technique may discard fault-revealing tests [49].

Using a safe regression test selection technique has the same fault detection effectiveness as the retest-all technique. All of the faults detectable by the entire test suite are detectable by the selected safe test suite subset.

2.4.2.2 Techniques found in the literature

Table 2-1 lists regression test selection techniques that have been applied to procedural code as found in the literature.

Table 2-2 lists regression test selection techniques that have been applied to object-oriented code as found in the literature.

Table 2-1 – Regression Test Selection Techniques for Procedural Code.

Experimental Technique	Description
Retest-all	Control technique. No selection technique is used. This technique is used in many studies (e.g., [15, 21, 37, 46, and 47]) as an experimental control.
Random	An unsafe technique. Randomly selects n% of the test cases from a test suite. This simple technique is used in studies (e.g., [21, 32]) for comparison against more complex techniques.
Modified entity	A safe technique. Chen et al. introduced the TestTube system in [8] to implement the modified entity technique. It associates each test in a test suite with the entities (i.e., functions, types, variables, and macros) in a software system. By using coverage and change information, the modified entity technique selects only the tests that exercise modified or deleted entities in *P'*. The modified entity technique has been evaluated in numerous subsequent studies (e.g., [3, 32, 46, and 47]). Tools: The TestTube system in [8] is implemented for C programs on the UNIX platform. It makes use of a number of existing analysis tools. Other studies (e.g., [46 and 47]) do not explicitly state that the TestTube system is used. These studies cite the use of a coverage tool to determine test coverage and control graph information; UNIX utilities and direct inspection are used to determine modified functions, or functions using modified structures.
Modified non-core entity	An unsafe technique. Used in multiple studies (e.g., [15, 37, 46, and 47]). Acts like the modified entity technique, but ignores core functions. A core function is defined as a function exercised by more than *k*% of the test cases in the test suite. The value *k* is determined by the experimenters. By having a *k* less than 100%, safety is sacrificed for efficiency. Were *k* to be 100%, selecting all test cases that exercise core functions might lead to no reduction in the size of *T* [47]. In [37, 46, and 47], *k* is set to 80%. In [15], *k* is set to 90%. Rothermel et al. in [46] indicate that *k* is chosen based on the results of previous studies of the technique's effectiveness. Tools: Implemented for C programs on the UNIX platform. The studies use a coverage tool to determine test coverage and control graph information; UNIX utilities and direct inspection are used to determine modified functions, or functions using modified structures.

Control-Flow-Based	A safe technique. Rothermel and Harrold introduce this technique in [48]. Constructs control flow graphs for a procedure or program as well as its modified version. The two sets of graph edges are then compared and analyzed to select the tests from the original program that can be used to execute the modified code. This approach has been evaluated in numerous subsequent studies (e.g., [3, 21, 32, and 50]). Tools: Implemented for C programs on the UNIX platform. The studies use a coverage tool to determine test coverage and control graph information. The DejaVu tool was created in [48] to implement the selection technique. (Due to limitations in the prototype analysis tools, the test selection has to be simulated in some cases.)
Modification-focused minimization	An unsafe technique. Fischer et al. present a technique in [20] that seeks to select a reduced set of test cases associated with the modified modules of a software system. This approach is used in [46] to attempt to select a minimal number of test cases required to exercise all the modified functions in *P'*. This technique is a heuristic; it uses the coverage information in applying *T* to *P* in order to predict the functions that will covered in *P'*. Tools: Implemented for C programs on the UNIX platform. Studies [15] and [46] use coverage tools to determine test coverage information; UNIX utilities and direct inspection are used to determine modified code. There is no mention of any specific software used or written to perform the minimization. Graves et al. in [21] created a simulator tool as a minimization technique to select a minimal test suite *T'* such that *T'* is edge-coverage-adequate for a set of edges, in the control-flow graphs for *P* or *P'*, that are associated with code modifications. Kim et al. [32] created a similar tool.
Dataflow-coverage-based	An unsafe technique. Selects test cases that exercise data interactions that have been affected by modifications. Dataflow techniques might be useful in detecting portions of P' that are not adequately tested [21]. Tools: Graves et al. in [21] simulated a tool by manually inspecting program modifications, and then generated a list of tuples that represented the definition-use (du) pairs that were affected by program modifications. A du pair was considered to be affected by a modification if it had been deleted from *P* in obtaining *P'*, or if it involved a definition or use contained in a statement or predicate that had been modified in creating *P'*. A set of selected test cases *T'* was then created by identifying the test cases in the original test suite *T*, that when executed on *P*, exercised the affected du pairs.
Ball's algorithm	A safe technique. Introduced by Ball in [2]. Builds on the work of Rothermel and Harrold with DejaVu in [48]. Also uses control flow graphs and their edges to analyze which tests to select from the original program. Ball offers additional algorithms that provide further precision based on control flow that are even more precise than the edge-based algorithms – be it at a greater computational cost.
Pythia	A safe technique. Introduced by Vokolos and Frankl in [63]. Selects test cases based on changes that have been made to a program. Test selection is made via textual differencing that can determine statement differences between two versions of a program. Tools: Pythia is a utility implemented for C programs on the UNIX platform.

Table 2-2 – Regression Test Selection Techniques for Object-Oriented Code.

Experimental Technique	Description
Retest-all	Control technique. No selection technique is used. This technique is used in many studies (e.g., [15, 21, 37, 46, and 47]) as an experimental control.
Control-flow-based (applied to C++ software)	A safe technique. The methodology expands upon the DejaVu methodology created in [48] for procedural software. Rothermel and Harrold examine object-oriented software testing in [51, 52]. In [51], a methodology for selecting regression tests based on both changes to control dependencies and data dependencies is presented. In [52], only changes to control flow are considered. Rothermel and Harrold [52] present a method of regression test selection for C++ software. Regression tests must be selected at the class testing level, for modified and newly derived classes. It is also desirable to retest all application program(s) using a modified class, thus regression test selection at the application level is also used. Application level testing: Interprocedural control flow graphs (ICFGs) are created to encode control flow for a group of interacting methods that have a single entry point, such as a group of methods that make up an entire program. Regression tests are selected by comparing an original ICFG with the ICFG of the subsequent version of a program. Class level testing: Typically, to perform class testing, a class driver is required. The driver must perform setup chores such as calling constructors, then invoke a sequence of methods, and then finally invoke an oracle to ascertain whether a class object has attained the expected state. When a class is modified, testers wish to identify which class tests should be rerun. Similarly, when a new class is derived from a base class, testers wish to identify which tests from the base class should be rerun on the derived class. For each class, a class control flow graph (CCFG) is constructed. A CCFG is a collection of individual control flow graphs for the methods in a class. CCFGs are used to select regression tests for classes. Similarly to the ICFG technique, regression tests are selected by comparing an original CCFG with the CCFG of the subsequent version of a class. Tools: Rothermel et al. in [52] simulated a tool for C++ programs on the UNIX platform.
Control-flow-based (applied to Java software)	A safe technique. Tool for regression testing Java software. Harrold et al. in [25] apply the methodology of the control-flow-based approach in [52] to Java software. Orso et al. in [43] further refine the approach to make it scaleable for testing large systems. Tools: DejaVOO [25] and other prototypical tools are used to perform the Java analyses.
Firewall approach for testing OO software	A safe technique. Kung et al. in [33] present a technique that constructs a firewall to enclose the set of classes affected by software changes and those that might be affected by the changes. These classes are the only ones that need to be retested. This type of testing only selects test cases at the class level. All the tests associated with testing a particular modified class are rerun, which is not very efficient. Builds upon the concept of Leung and White in [35] who introduced the concept of firewalls at the module level for procedural languages.

2.4.2.3 *x*Suds supported techniques

Table 2-3 presents an overview of the regression test selection techniques supported by *x*Suds.

Table 2-3 – Regression Test Selection Techniques Supported by *x*Suds.

Experimental Technique	Description
Modification-Based Regression Test Selection	Wong et al. present the technique in [66]. The *x*Suds manual [61] describes how the technique works. Execution trace files of *T* on *P* are stored. Regression tests *T'* on *P'* are selected by selecting regression tests that execute code that has been deleted or changed. Tests are selected for code that has been added as follows: tests are selected that execute the line of code that is just before and after corresponding position of the added code in the new release. The code coverage criteria can be specified. Table 2-9 lists the various code coverage options available.
Modification-Focused Minimization	Wong et al. present the technique in [66]. The *x*Suds manual [61] further describes how the technique works. Only the modified code is considered for the test cases. A minimization is then performed against the test cases based on their code coverage of the modified code. The code coverage criteria can be specified. Table 2-9 lists the various code coverage options available. (See the description for the coverage-focused minimization in Table 2-5 for more information on how the minimization is performed.)

2.4.3 Test suite reduction

As a software program evolves, its corresponding test suite will be modified to accommodate new functionality. As the test suite *T* grows, it is possible that redundant test cases will be introduced. The test suite reduction technique addresses this problem by seeking to permanently remove redundant test cases in *T*. The goal is to create a *T'* that is more efficient. The reduction process is typically accomplished without using prior knowledge of *P'* [46].

Rothermel et al. in [46] demonstrate that the use of a minimization technique significantly reduces execution time while maintaining a high level of fault detection effectiveness when compared to the retest-all approach.

Other studies have shown that test suite minimization can reduce the size of a test suite but that the fault detection effectiveness decreases with increasing test-suite sizes [23]. Wong et al. in [66] conducted a series of minimization experiments in which the resulting

loss in fault detection due to minimization is not very significant. Rothermel et al. in [53] conducted their own set of minimization experiments that contradict the previous study's findings; specifically, the loss of fault detection capability was found to be very significant.

2.4.3.1 Techniques found in the literature

Table 2-4 lists regression test suite reduction techniques as found in the literature that have been applied to procedural code.

Table 2-4 – Test Suite Reduction Techniques.

Experimental Technique	Description
No reduction	Control technique. No reduction technique is used. This technique is used in many studies (e.g., [37, 46, and 47]) as an experimental control.
Random	Control technique. Randomly selects n% of the test cases from a test suite. This simple technique is used in studies (e.g., [54, 67]) for comparison against more complex techniques.
Gupta-Harrold-Soffa (GHS) reduction	Harrold et al. introduce this technique in [24]. The technique is analysed in many subsequent studies (e.g., [15, 37, 46, 47, 53, and 54]). Attempts to minimize a test suite for a given coverage criterion. The reduction is accomplished by identifying, then eliminating redundant and obsolete test cases. The reduction technique is independent of the test methodology used. The technique can be used as long as an association between requirements and test cases can be made. Given: A test suite *TS*, a set of test case requirements r_1, r_2, …, r_n that must be satisfied to provide the desired testing coverage of the program, and subsets of *TS*, T_1, T_2,…, T_n, one associated with each of the r_i's such that any one of the test cases t_j belonging to T_i can be used to test r_i. GHS reduction is a heuristic that attempts to minimize the test suite while still meeting the test case requirements. That is, a minimal set of test cases from *TS* is found to satisfy all of the r_i requirements. In [24], a data flow coverage test approach was used as an example methodology for the GHS reduction. Tools: In the various studies, a UNIX based coverage tool was used to obtain test coverage information. The GHS algorithm is implemented as part of the test infrastructure.
Offutt et al's coverage based test reduction	Introduced by Offut et al. in [41]. A heuristic based approach that reduces test suite size by reordering the test case execution sequence. Test cases are generally executed in the sequence in which they have been generated. This technique works by minimizing the test set by employing a strategy to vary the order of the test cases. By reordering the test set, it is shown how a coverage criterion could be met by using a test set a fraction of the original's size. In [41] the technique is demonstrated using mutation and statement coverage.

2.4.3.2 *x*Suds supported techniques

Table 2-5 presents an overview of the test reduction technique supported by *x*Suds.

Table 2-5 – Test Suite Reduction Techniques Supported by *x*Suds.

Experimental Technique	Description
Coverage-Focused Minimization	Wong et al. in [66] describe the minimization as "an implicit enumeration algorithm with reductions to find the optimal subset based on all tests examined." This is a heuristic-based minimization tool. The algorithm is proprietary but the concept is similar to the GHS reduction approach. Minimizes a test set such that a minimal subset is found that preserves code coverage with respect to the specified code coverage criteria. Table 2-9 lists the various code coverage options available.

2.4.4 Prioritization test techniques

Prioritization test techniques seek to schedule test cases such that those with higher priority, according to some criterion, are run earlier in the test phase than others of lower priority. Rothermel et al. in [55] describe several goals of prioritization; one such goal is to increase "the likelihood of revealing faults earlier in the testing process." Informally they describe this goal as one of improving the test suite's rate of fault detection.

2.4.4.1 General test case prioritization vs. version-specific prioritization

Elbaum et al. outline two varieties of test case prioritization in [17].

In general test case prioritization, given a program *P* and a test suite *T*, the test cases in *T* are prioritized with the goal of finding an ordering of test cases that will be useful over a succession of the future modified versions of *P*. The desire is to have a prioritized test suite that is more successful in meeting the goals of the prioritization than the original test suite on average over the subsequent releases.

In version-specific prioritization, given a program *P* and a test suite *T*, the test cases in *T* are prioritized such that the modified ordering of *T* will be useful for a specific version *P'* of *P*. This prioritization is performed after changes have been made to *P* but prior to regression testing *P'*. The prioritized test suite may be more effective at regression testing version *P'* in particular than would be the case had the general test case prioritization approach been applied, but may be less effective on average over a

succession of subsequent releases.

2.4.4.2 Metrics associated with prioritizations

Some prioritization techniques make use of metrics in order to determine the test order. Two such metrics are the fault-exposing-potential and the fault-index.

2.4.4.2.1 Fault-exposing-potential (FEP)

FEP is an approximation of whether a fault in a statement (or function, etc.) will cause a failure for a test case executing it. Ranking the test cases according to their corresponding FEP values has been proposed as a prioritization criterion.

Elbaum et al. in [17] present an algorithm to approximate the statement-level FEP of a test case that is based on mutation analysis: Given a program P and a test suite T, for each test case $t \in T$, for each statement s in P, determine the mutation score $ms(s, t)$ of t on s to be the ratio of mutants of s exposed by t to total mutants of s. Then calculate, for each test case t_k in T, an award value for t_k, by summing all $ms(s, t_k)$ values. These award values approximate the FEP values of each of the test cases.

Chen et al. in [7] examine the efficacy of the FEP measure in practice. Though they conclude that using the FEP measure in prioritization calculations yielded improved performance, the increase in performance was actually quite small (i.e., small enough to question the value of employing the technique).

2.4.4.2.2 Fault-index (FI)

FI is a metric of fault proneness, defined in [18], that indicates the likelihood that a fault exists in a function or some other code entity of interest (e.g., a statement, etc.). The FI value can be associated with attributes of the software under study; as code changes are a primary focus of regression testing, the FI value can also take into account the modifications made to the software. The exact nature of the changes will impact the value. Using FI values to prioritize test cases has also been proposed as a prioritization criterion.

An algorithm to estimate the FI of a given function is as follows: "Given program P and

subsequent version *P'*, generating (regression) fault indexes for *P'* requires generation of a fault index for each function in *P*, generation of a fault index for each function in *P'*, and a function-by-function comparison of the indexes for *P'* against those calculated for *P*. As a result of this process, the regression fault proneness of each function in *P'* is represented by a regression fault index based on the complexity of the changes that were introduced into that function."

Given the fault indexes, and the coverage information for each test case in *T*, an association between the fault indexes can be made to each individual test case. For example, one might choose to sum up the fault indexes associated with the functions exercised by each test case and then prioritize the test cases according to these summed fault index values.

2.4.4.3 Summary of the techniques found in the literature

As with regression test selection, most of the existing studies focus on procedural languages (e.g., [14, 16, 17, 18, 19, 55, and 56]); though other studies (e.g., [11, 12]) examine these techniques with respect to testing the Java language. These studies make use of software tools to acquire the necessary data to perform the prioritizations. All of the non-control techniques require that code coverage information be recorded for each test case. Other techniques require code modification information, code metrics to implement techniques involving the FI measure, or the calculation of mutant scores (obtained via mutant analysis) to implement the FEP related techniques.

Table 2-6 lists test suite prioritization techniques as found in the literature that have been applied to procedural code. Table 2-7 lists test suite prioritization techniques as found in the literature that have been applied to object-oriented code.

Table 2-6 – Test Suite Prioritization Techniques for Procedural Code.

Technique	Description
No prioritization	Control technique. Maintain the test suite's original ordering. Allows for the consideration of untreated test suites. Evaluated in studies [55 and 56].
Random prioritization	Control technique. Randomly order the test cases in a test suite. Evaluated in studies [15, 16, 17, 18, 19, 47, 55, and 56].

Optimal prioritization	Control technique. This "technique" is only possible given a program that contains known faults. For any test suite, given the knowledge of which test cases expose which faults, it is possible to create an optimal ordering of test cases for maximizing that suite's rate of fault detection. This is not a viable practical technique, but it provides an upper bound on the effectiveness of any prioritization heuristics being evaluated in empirical studies. By comparing the results of a viable technique to this one we can gage the effectiveness of the technique under study. Evaluated in studies [15, 17, 18, 37, 46, 47, 55, and 56].
Total statement coverage prioritization	Prioritize test cases according to the total number of program statements they cover. Evaluated in studies [17, 18, 55, and 56].
Additional statement coverage prioritization	Combines feedback with coverage information. It iteratively selects a test case that yields the greatest statement coverage, adjusts the coverage information on subsequent test cases to indicate their coverage of statements not yet covered, and repeats this process until all statements covered by at least one test case have been covered. If multiple test cases cover the same number of statements not yet covered, they are ordered randomly. When all statements have been covered, this process is repeated on the remaining test cases until all have been ordered. Evaluated in studies [16, 17, 18, 55, and 56].
Total statement fault-exposing-potential prioritization	Prioritize test cases in order of the probability of exposing faults at the statement level. The fault-exposing-potential (FEP) technique is used. Evaluated in studies [17, 18, 55, and 56].
Additional statement fault-exposing-potential prioritization	Prioritize test cases in order of the probability of exposing faults at the statement level, adjusted to consider the previous test cases. The fault-exposing-potential (FEP) technique is used. Evaluated in studies [17, 18, 55, and 56].
Total branch coverage prioritization	Same as total statement coverage prioritization, except that it relies on coverage measured in terms of program branches. Evaluated in studies [55, 56].
Additional branch coverage prioritization	Same as additional statement coverage prioritization, except that it relies on coverage measured in terms of program branches. Evaluated in studies [55, 56].
Total function coverage prioritization	Same as total statement coverage prioritization, except that it relies on coverage measured in terms of program functions. Evaluated in studies [15, 17, 18, 19, 37, and 46],
Additional function coverage prioritization	Same as additional statement coverage prioritization, except that it relies on coverage measured in terms of program functions. Evaluated in studies [15, 16, 17, 18, 19, 37, 46, and 47].
Total function diff prioritization	Uses modification information. It sorts test cases in the order of their coverage of functions that have changed. If multiple test cases cover the same number of differing functions, they are ordered randomly. Total function coverage prioritization is applied to the remaining test cases. Evaluated in studies [18, 19].
Additional function diff prioritization	Uses both feedback and modification information. It iteratively selects a test case that yields the greatest coverage of functions that differ, adjusts the information on subsequent test cases to indicate their coverage of functions not yet covered, and then repeats this process until all functions that differ and have been covered by at least one test case have been covered. If multiple test cases cover the same number of differing functions not yet covered, they are ordered randomly. This process is repeated until all

	test cases that execute functions that differ have been used. Additional function coverage prioritization is applied to remaining test cases. Evaluated in studies [18, 19].
Total function fault-exposing-potential prioritization	Prioritize test cases in order of the probability of exposing faults at the function level. The fault-exposing-potential (FEP) technique is used. Evaluated in studies [17, 18].
Additional function fault-exposing-potential prioritization	Prioritize test cases in order of the probability of exposing faults at the function level, adjusted to consider the previous test cases. The fault-exposing-potential (FEP) technique is used. Evaluated in studies [17, 18].
Total function fault-index prioritization	Prioritize test cases based on the probability of fault existence at the function level. The fault-index (FI) technique is used. Evaluated in studies [17, 18].
Additional function fault-index prioritization	Prioritize test cases based on the probability of fault existence at the function level, adjusted to consider the previous test cases. The fault-index (FI) technique is used. Evaluated in studies [16, 17, and 18].
Total function fault-index with fault-exposing-potential prioritization	Prioritize test cases based on the combined probabilities of fault existence and fault exposure. Both the FEP and FI techniques are used. Evaluated in studies [17, 18].
Additional function fault-index with fault-exposing-potential prioritization	Prioritize test cases based on the combined probabilities of fault existence and fault exposure, adjusted to consider previous test cases. Both the FEP and FI techniques are used. Evaluated in studies [17, 18].
Total function diff with fault-exposing-potential prioritization	Prioritize test cases based on the combined probabilities of fault existence and fault exposure (based on modification information). Both modification-information and FEP techniques are used. Evaluated in study [18].
Additional functional diff with fault-exposing-potential prioritization	Prioritize test cases based on the combined probabilities of fault existence and fault exposure (based on modification information), adjusted to consider previous test cases. Both modification-information and FEP techniques are used. Evaluated in study [18].

Table 2-7 – Test Suite Prioritization Techniques for Object-Oriented Code.

Technique	Description
No prioritization	Control technique. Maintain the test suite's original ordering. Allows for the consideration of untreated test suites. Evaluated in study [12].
Random prioritization	Control technique. Randomly order the test cases in a test suite. Evaluated in study [12].

Optimal prioritization	Control technique. This "technique" is only possible given a program that contains known faults. For any test suite, given the knowledge of which test cases expose which faults, it is possible to create an optimal ordering of test cases for maximizing that suite's rate of fault detection. This is not a viable practical technique, but it provides an upper bound on the effectiveness of any prioritization heuristics being evaluated in empirical studies. Evaluated in study [12].
Total block coverage prioritization	Prioritize test cases according to the total number of program basic blocks they cover. Evaluated in study [12].
Additional block coverage prioritization	Combines feedback with coverage information. It iteratively selects a test case that yields the greatest block coverage, adjusts the coverage information on subsequent test cases to indicate their coverage of blocks not yet covered, and repeats this process until all blocks covered by at least one test case have been covered. If multiple test cases cover the same number of blocks not yet covered, they are ordered randomly. When all blocks have been covered, this process is repeated on the remaining test cases until all have been ordered. Evaluated in study [12].
Total method coverage prioritization	Same as total block coverage prioritization, except that it relies on coverage measured in terms of methods. Evaluated in study [12].
Additional method coverage prioritization	Same as additional block coverage prioritization, except that it relies on coverage measured in terms of methods. Evaluated in study [12].
Total diff method prioritization	Uses modification information. It sorts test cases in the order of their coverage of methods that have changed. If multiple test cases cover the same number of differing methods, they are ordered randomly. Total method coverage prioritization is applied to any remaining test cases. Evaluated in study [12].
Additional diff method prioritization	Uses both feedback and modification information. It iteratively selects a test case that yields the greatest coverage of methods that differ, adjusts the information on subsequent test cases to indicate their coverage of methods not yet covered, and then repeats this process until all methods that differ and have been covered by at least one test case have been covered. If multiple test cases cover the same number of differing methods not yet covered, they are ordered randomly. This process is repeated until all test cases that execute methods that differ have been used; additional method coverage prioritization is applied to any remaining test cases. Evaluated in study [12].
Echelon/Scout [60]	Microsoft has developed an in-house test system named Scout (initially called Echelon). It is a prioritization tool that acts on compiled code at the binary block level. It uses a "fast, simple and intuitive" heuristic that computes which tests will cover affected basic blocks of program. New block and modified blocks of programs are prioritized. The same approach as with additional diff method prioritization is used, except that it relies on coverage measured in terms of binary blocks.

2.4.4.4 *x*Suds supported techniques

Table 2-8 lists all of the prioritizations performed for this experiment. (Table 2-9 provides the definitions of the specified code coverage criteria used for the

prioritizations.)

*x*Suds provides the necessary code coverage information related to each test case to perform the *total-coverage* prioritizations. *x*Suds also provides a software utility to execute the *additional-coverage* prioritizations. For both the *total* and *additional* coverage approaches the option to prioritize based on only the modified code can also be achieved with additional code analysis using the *x*Suds tool suite.

Table 2-8 – Test Suite Prioritization Techniques Supported by *x*Suds.

Label	Mnemonic	Description
Control techniques.		
M1	untreated	Maintain the test suite's original ordering.
M2	random	Randomly order the test cases in a test suite.
M3	optimal	This "technique" is only possible given a program that contains known faults. For any test suite, given the knowledge of which test cases expose which faults, it is possible to create an optimal ordering of test cases for maximizing that suite's rate of fault detection.
Total-coverage techniques.		
M4	function-entry-total	Prioritize test cases according to the total number of function entries. If multiple test cases cover the same number of function entries, they are ordered randomly.
M5	function-return-total	Same as function-entry-total coverage prioritization, except that it relies on coverage measured in terms of function returns.
M6	block-total	Same as function-entry-total coverage prioritization, except that it relies on coverage measured in terms of blocks.
M7	basic-block-total	Same as function-entry-total coverage prioritization, except that it relies on coverage measured in terms of basic-blocks.
M8	decision-total	Same as function-entry-total coverage prioritization, except that it relies on coverage measured in terms of decisions.
Additional-coverage techniques.		
M9	function-entry-addtl	Combines feedback with coverage information. It iteratively selects a test case that yields the greatest function entry coverage, adjusts the coverage information on subsequent test cases to indicate their coverage of function entries not yet covered, and repeats this process until all function entries covered by at least one test case have been covered. If multiple test cases cover the same number of function entries not yet covered, then one of these test cases is selected randomly. When all function entries have been covered, any remaining test cases must still be ordered. This is done by resetting all function entries to the status of "not covered"; function-entry-addtl prioritization is then repeated on the remaining test cases until they all have been ordered.
M10	function-return-addtl	Same as function-entry-addtl coverage prioritization, except that it relies on coverage measured in terms of function returns.
M11	block-addtl	Same as function-entry-addtl coverage prioritization, except that it relies on coverage measured in terms of blocks.
M12	basic-block-addtl	Same as function-entry-addtl coverage prioritization, except that it relies on coverage measured in terms of basic-blocks.
M13	decision-addtl	Same as function-entry-addtl coverage prioritization, except that it relies on coverage measured in terms of decisions.

Total-coverage-using-modification-information techniques.		
M14	function-entry-diff-total	Uses modification information. It sorts test cases in the order of their coverage of function entries that have changed from P to P'. If multiple test cases cover the same number of differing function entries, they are ordered randomly. Function-entry-total prioritization is applied to any test cases that remain unsorted by this technique.
M15	function-return-diff-total	Same as function-entry-diff-total prioritization, except that it relies on coverage measured in terms of function returns.
M16	block-diff-total	Same as function-entry-diff-total prioritization, except that it relies on coverage measured in terms of blocks.
M17	basic-block-diff-total	Same as function-entry-diff-total prioritization, except that it relies on coverage measured in terms of basic-blocks.
M18	decision-diff-total	Same as function-entry-diff-total prioritization, except that it relies on coverage measured in terms of decisions.
Additional-coverage-using-modification-information techniques.		
M19	function-entry-diff-addtl	Uses both feedback and modification information. It iteratively selects a test case that yields the greatest coverage of function entries that differ from P to P', adjusts the coverage information on subsequent test cases to indicate their coverage of differing function entries not yet covered, and then repeats this process until all function entries that differ and have been covered by at least one test case have been covered. If multiple test cases cover the same number of differing function entries not yet covered, then one of these test cases is selected randomly. When all differing function entries have been covered, any remaining test cases must still be ordered. This is done by resetting all differing function entries to the status of "not covered"; function-entry-diff-addtl is then repeated on the remaining test cases. Function-entry-addtl prioritization is applied to any test cases that remain unsorted by this technique.
M20	function-return-diff-addtl	Same as function-entry-diff-addtl prioritization, except that it relies on coverage measured in terms of function returns.
M21	block-diff-addtl	Same as function-entry-diff-addtl prioritization, except that it relies on coverage measured in terms of blocks.
M22	basic-block-diff-addtl	Same as function-entry-diff-addtl prioritization, except that it relies on coverage measured in terms of basic-blocks.
M23	decision-diff-addtl	Same as function-entry-diff-addtl prioritization, except that it relies on coverage measured in terms of decisions.

2.5 Test Suite Granularity

Rothermel et al. in [47] examine the impact of test suite granularity on the cost-effectiveness of regression testing. A "test grain" is defined as the smallest input that can be used as a test case. A test case can be comprised of one or a few test grains (resulting in a finer grained test case) or many test grains (resulting in a coarser grained test case). The study finds that coarser grained testing can greatly increase the efficiency of a test-suite due to a reduction in testing overhead. However, fine granularity is more supportive

of regression test selection and test suite reduction. The effectiveness of these regression testing techniques decreases as granularity becomes coarser. Also of interest in the study is the fact that fine granularity tests were found to have a lower level of fault detection than did coarser granularity ones. A hypothesis to explain this occurrence is that the coarser grained tests caused additional code to be executed, that in turn caused data state changes occurring in earlier stages of execution to be visible. A possible conclusion to be drawn is that the use of a more precise oracle might be required in order to create more effective fine grained test cases.

2.6 Additional Factors

Elbaum et al. present a study in [13] to gauge the impact of software evolution on code coverage data. The study suggests that even small modifications made to the software can affect code coverage information, and the degree of the impact of change on coverage can be difficult to predict. This study suggests that when test cases are initially written for some software and code coverage information is taken, code coverage across subsequent versions may not be stable.

Harrold in [23] suggests that using a particular regression testing technique is not the only factor affecting the performance of a regression test suite; other factors influencing outcomes include the program used, the types of modifications made to it, and the composition of a test suite.

2.7 Measuring Regression Test Effectiveness

2.7.1 Leung and White cost-effectiveness model

Leung and White present a test cost model in [34], which identifies the conditions in which a selection strategy is more economical than the retest-all option. The model assumes that both retest-all and selection strategies give the same test effectiveness. Thus, this model is really only appropriate for evaluating safe selection techniques, as the technique does not consider the cost of missing faults due to discarded tests.

The following costs can be attributed to the activities involved in testing a software system:

1) System Analysis Cost, *Ca*. The costs associated with becoming familiar with a system (i.e., examining specifications, design, and possibly the program implementation). In general, the larger the system, the higher the cost.

2) Test Selection Cost, *Cs*. After gaining knowledge of a system, the test analyst can select the test cases appropriate for testing its behaviour. The cost depends largely on the test strategy selected.

3) Test Execution Cost, *Ce*. Cost of setting up the environment necessary to execute tests (e.g., assembling a laboratory full of the hardware necessary to test a computer networking solution).

4) Result Analysis Cost, *Cr*. Cost associated with checking the resulting behaviour of a system against the expected results. The cost *Cr* can further be reduced to two subcomponents, *Cu* and *Cc*. The cost *Cu* represents the cost of understanding the program and specifications to determine whether the resulting behaviour is correct. The cost *Cc* represents the cost of checking the resulting output to its expected value.

The following equation represents the cost of a test strategy S that uses a test set T:

$$C(S) = Ca(T) + Cs(T) + Ce(T) + Cu(T) + Cc(T)$$

Let *To* represent a set of old tests and *Tn* represent a set of new tests. The cost of a retest-all regression strategy is given as follows:

$$C(retest\text{-}all) = \begin{matrix}[Ce(To) + Cu(To) + Cc(To)] \\ +[Ca(Tn) + Cs(Tn) + Ce(Tn) + Cu(Tn) + Cc(Tn)]\end{matrix}$$

$Ca(To) = Cs(To) = 0$ as the retest-all strategy does not analyze previous test sets before applying them and no effort is needed to select these tests.

The cost of a selective regression strategy that uses a subset test set *Ts* of *To* together with new tests *Tn* (it is assumed that the same strategy is used to determine *Tn* in both selective and retest-all approaches) is given as follows:

$$C(selective) = \begin{matrix}[Ca(Ts) + Cs(Ts) + Ce(Ts) + Cu(Ts) + Cc(Ts)] \\ + [Ca(Tn) + Cs(Tn) + Ce(Tn) + Cu(Tn) + Cc(Tn)]\end{matrix}$$

The cost of a selective strategy is less than the retest-all strategy if:

$$C(selective) - C(retest\text{-}all) < 0$$

or

$$[Cs(Ts) + Ce(Ts) + Cc(Ts)] - [Ce(To) + Cc(To)] < 0$$

since one can hypothesize that $Ca(Ts) + Cu(Ts) \approx Cu(To)$; in both selective and retest-all cases the test analyst has to understand the effects of the modifications on the validity of the previous test cases.

Rearranging the previous equation results in:

$$Cs(Ts) < [Ce(To) - Ce(Ts)] + [Cc(To) - Cc(Ts)]$$

The selective strategy is cost-effective if the cost for selecting a subset of the previous tests is less than the cost for executing and checking the extra previous tests needed for the retest-all strategy.

Assume that *Cs(T)*, *Ce(T)*, and *Cc(T)* are directly proportional to the number of tests in *T*. Let $|T|$ represent the cardinality of test set *T*. Let *s*, *e*, and *c* be constants representing the cost for selection cost, execution cost, and checking cost respectively. Then the previous inequality can be rewritten as:

$$s|Ts| < e(|To| - |Ts|) + c(|To| - |Ts|) = (e + c)(|To| - |Ts|)$$

or

$$s < (e + c)\left(\frac{|To|}{|Ts|} - 1\right)$$

Provided the above inequality holds, then the selective test strategy is more economical

than the retest-all strategy.

Of course, this model makes assumptions that s, e, and c are constant for all test cases. It is also assumed that all costs are quantifiable. These assumptions are not generally realistic.

2.7.2 Rosenblum and Weyuker predictors of cost-effectiveness

Rosenblum and Weyuker in [45] have proposed a technique to generate predictors of the cost-effectiveness of regression testing. In considering cost, they consider two factors: the cost of executing the test cases and the cost of the computations involved in any selection process. They considered two main classes of selective regression testing: safe regression selection, and minimization. For their purposes, Rosenblum and Weyuker assume that minimization is unlikely to cause faults to go undetected when compared to the retest-all approach.

Let P denote a system under test and let T denote the regression test suite for P, with $|T|$ denoting the number of individual test cases in T. Let M be the selective regression test method used to choose a subset of T for testing a modified version of P and let E be the set of entities of the system under test that are considered by M. It is assumed that T and E are nonempty and that every syntactic element of P belongs to at least one entity in E. Let E^C denote the set of covered entities, with $|E^C|$ representing the number of covered entities.

Let $C_{i,j}$ represent an element of a matrix C whose rows represent elements of T and whose columns represent elements of E. Element $C_{i,j}$ is defined to be one if and only if the execution of P on test case t causes entity e to be exercised at least once, it is zero otherwise. Let CC represent the *cumulative coverage* achieved by T:

$$CC = \sum_{i=1}^{|T|} \sum_{j=1}^{|E^C|} C_{i,j}$$

Recall that Leung and White have shown that to be cost-effective:

$$s|Ts| <= (e+c)(|To|-|Ts|) = r(|To|-|Ts|)$$

Note that the variable r is introduced to be the combined execution and checking cost.

To determine whether regression testing technique M is cost-effective when used for software P, we must first consider whether it is cost-effective when only a single entity of P is changed.

To determine whether M is cost-effective for testing a single change to P, we compute the average number of test cases that cover each covered entity:

$$N_M^C = \frac{CC}{|E^C|}$$

The fraction of the test suite that needs to be rerun is π_M, which is a predictor for $|T_M|/|T|$:

$$\pi_M = \frac{N_M^C}{|T|} = \frac{CC}{|E^C||T|}$$

Thus, if a change is made to one typical (and we assume a covered) entity, then N_M^C test cases (or π_M of T) will be needed to test the change.

The same coverage analysis that would be used in performing M must be used to compute π_M. When performing this coverage analysis, the cost of the analysis can be computed, as well as the cost r of running a test case. This gives a lower bound on s_M, the per-test-case cost of applying M; the other component of the cost is the cost of the change analysis. The estimate of the per-test-case coverage analysis cost is called $s_{M,C}$. Once π_M and $s_{M,C}$ have been computed, they can be plugged into a model such as the Leung-White cost model, and along with r can be used to determine whether or not M is cost-effective for the selection rate π_M. If it is, then a more accurate estimate must be computed for multiple changed entities as this is typically the case.

Rosenblum and Weyuker go on to demonstrate how N_M^C and π_M can be calculated for safe strategies, with multiple entities changed. They provide some simple examples for different coverage patterns (e.g., sometimes multiple entities are covered by many of the same tests; other times there might be fewer shared tests).

The minimization technique is similarly examined. In the case where a single entity is changed, only one test needs to be rerun. However, where multiple entities are involved, Rosenblum and Weyuker again demonstrate how N_M^C and π_M can be calculated for different coverage patterns.

The simple code coverage patterns offered as examples are not necessarily indicative of the real-world, as more sophisticated methods are required.

Harrold et al. revisit the Rosenblum and Weyuker predictor in [26]. They propose that not only must the predictor account for code coverage, it must also account for the modification distribution. They attempt to improve the predictor by adding weights that represent the relative frequency of changes to the covered entities. The weights serve to incorporate information on the distribution of modifications made from one version to another. Weighted analogs of N_M^C and π_M were created as follows:

$$WN_M^C = \sum_{j=1}^{|E^C|} w_j \sum_{i=1}^{T} C_{i,j}$$

and

$$\Pi_M = \frac{WN_M^C}{|T|}$$

2.7.3 Kim et al.'s cost measures

A problem with the Leung and White approach is that it does not take into account the cost of undetected faults. Kim et al. present an alternative model to the Leung and White model in [32]. This alternative assumes that the test selection is cost-effective by Leung and White's definition; the model examines the costs incurred when the selected test

cases do not detect the faults that could have been detected by the original test set.

Two measures of effectiveness that are fault-focused are proposed. An analysis is performed on the output of P and P' to identify whether the output differs for each test in T. Let NF_{det} be the number of faults for which there is at least one fault-revealing test case in T. Similarly, let NF_{det}' be the number of faults for which there is at least one fault-revealing test case in T'. Finally, let NF be the total number of faults in P'.

The following two metrics, *relative effectiveness* and *absolute effectiveness*, are proposed:

$$Relative\ Effectiveness = \frac{NF_{det}'}{NF_{det}}$$

and

$$Absolute\ Effectiveness = \frac{NF_{det}'}{NF}$$

2.7.4 Rothermel et al.'s savings and cost measures

Rothermel et al. proposed metrics in [53] to evaluate the costs and benefits of minimizing a test suite.

2.7.4.1 Measuring savings

Test suite minimization allows for potentially dramatic reductions in testing times. The savings in time are mostly dependent on the number of tests that can be removed from a retesting effort. Thus, a measure representing the percentage reduction in test suite size resulting from minimization can represent the savings, and is given by the following:

$$Percentage\ Reduction\ in\ Test\ Suite\ Size = \frac{|T| - |T_{\min}|}{|T|} \times 100$$

This is a simple approach that makes several assumptions, including that all test cases are equal in cost (e.g., requiring equal human or equipment times), and it does not address

the compounding of savings that results from minimization over a succession of subsequent releases.

2.7.4.2 Measuring costs

Two costs are addressed that play a role in test set minimization. The first is the cost of executing a minimization tool. An assumption is made that the cost of running the tool is non-critical as it can be run in off-peak hours, automatically, and/or after the release of a product. Also, the cost of minimization might be amortized over the release of subsequent versions of the product thereby assuming a less significant role in the overall cost. The second, more significant, cost is the cost associated with discarding tests that, if executed, would reveal defects in the software. The cost of the reduced effectiveness may be compounded over subsequent product releases. Also, the effects of the missed faults might be critical.

Let $|F|$ represent the number of faults revealed by test set T over the faulty versions of a program P. Let $|F_{min}|$ denote the number of faults revealed by T_{min} over those versions of P. A measure representing the percentage reduction in fault detection effectiveness for the minimization of T for P is given by the following:

$$\textit{Percentage Reduction in Fault Detection Effectiveness} = \frac{|F| - |F_{\min}|}{|F|} \times 100$$

This method of measuring the costs associated with minimization calculates cost relative to a fixed set of faults. The method also assumes that the costs associated with missed faults are equal.

2.7.5 Malishevsky et al.'s cost models

Malishevsky et al. present cost-benefit models in [37] for regression test selection, test suite reduction, and test case prioritization.

2.7.5.1 Regression test selection

This model builds on Leung and White's model presented above. The following

variables are introduced:

1) Cost of analysis, *Ca(T)*. Includes the cost of source code analysis, analysis of changes between versions, and the collection of execution traces.

2) Cost of execution, *Ce(T)*. Includes the cost of preparing a program and of running the tests from *T*.

3) Cost of result checking, *Cc(T)*. Includes the cost of validation of program outputs either automatically with previously saved results, or by manual inspection.

4) Cost of selection, *Cs(T)*. Includes the cost of running a test selection tool.

5) Cost of maintenance of the test suite, *Cm(T)*. Includes the cost of maintaining a database of tests and program outputs from previous executions.

6) Cost of omitting faults by not selecting $T \setminus T'$, $Cf(F(T) \setminus F(T'))$. Includes the costs of missing faults such as lower customer satisfaction with a product, litigation, or the cost of releasing product updates to fix the faults.

The cost of the retest-all strategy is given by:

$$C = Ce(T) + Cc(T) + Cm(T)$$

The cost of a regression test selection strategy is:

$$C' = Ca(T) + Ce(T') + Cc(T') + Cs(T) + Cf(F(T) \setminus F(T')) + Cm(T)$$

The regression test selection strategy is more cost effective than retest-all if $C' < C$. Substituting the above expressions results in:

$$Ca(T) + Ce(T') + Cc(T') + Cs(T) + Cf(F(T) \setminus F(T')) < Ce(T) + Cc(T)$$

This model can also be used to compare the cost-effectiveness of two selection techniques. Let T' be the test suite selected by technique 1 and let T'' be the test suite selected by technique 2. The corresponding regression testing costs are:

$$C_1 = Ca_1(T) + Ce(T') + Cc(T') + Cs_1(T) + Cf(F(T) \setminus F(T'))$$

and

$$C_2 = Ca_2(T) + Ce(T'') + Cc(T'') + Cs_2(T) + Cf(F(T) \setminus F(T''))$$

when techniques 1 and 2 are employed, respectively. Analysis costs might differ as different data might be required and selection costs might differ because different techniques may use different selection algorithms. The technique with the lower C_k is the most cost beneficial.

2.7.5.2 Test suite reduction

When constructing a cost model for reduction we assume that after test suite T is reduced, regression testing is performed g times. The variables *Ca(T)*, *Ce(T)*, *Cc(T)*, *Cs(T)*, and *Cm(T)*, are as defined for the regression test selection technique. The variable $Cf(F_k(T) \setminus F_k(T'))$ is defined to be the cost of omitting faults, where $F_k(x)$ is the set of regression faults on version v_k detected by a test suite x.

The cost of the retest-all strategy is given by:

$$C = g \times Ce(T) + g \times Cc(T) + g \times Cm(T)$$

The cost of a reduction strategy is:

$$C' = Ca(T) + g \times Ce(T') + g \times Cc(T') + Cs(T) + \sum_{1 \le k \le g} Cf(F_k(T) \setminus F_k(T')) + g \times Cm(T')$$

The reduction strategy is more cost effective than retest-all if $C' < C$. Substituting the above expressions yields:

$$Ca(T) + g \times Ce(T') + g \times Cc(T') + Cs(T) + \sum_{1 \le k \le g} Cf(F_k(T) \setminus F_k(T')) + g \times Cm(T') < g \times Ce(T) + g \times Cc(T) + g \times Cm(T)$$

This model can also be used to compare the cost-effectiveness of two reduction

techniques. Let T' be the test suite selected by technique 1 and let T'' be the test suite selected by technique 2. The corresponding regression testing costs are:

$$C_1 = Ca_1(T) + g \times Ce(T') + g \times Cc(T') + Cs_1(T) + \sum_{1 \le k \le g} Cf(F_k(T) \setminus F_k(T')) + g \times Cm(T')$$

and

$$C_2 = Ca_2(T) + g \times Ce(T'') + g \times Cc(T'') + Cs_2(T) + \sum_{1 \le k \le g} Cf(F_k(T) \setminus F_k(T'')) + g \times Cm(T'')$$

when techniques 1 and 2 are employed, respectively. As with selection, analysis costs and selection costs might differ. The technique with the lower C_k is the most cost beneficial.

2.7.5.3 Test case prioritization

Finally, this is the cost model for prioritization. Given a test suite T, the following variables are defined:

1) Cost of analysis, *Ca(T)*. Defined as above.

2) Cost of the prioritization algorithm, *Cp(T)*. The actual cost of running a prioritization tool.

We next define $delays^O$ as the cumulative cost of waiting for each fault to be exposed while executing test suite T under order O. Let m be the number of faults, e_k^O is the runtime and validation time of test k in suite T under order O, TF_i^O is the test number under order O that first detects fault i, and f_i the cost of waiting for fault i to be exposed (e.g., different programmers who wait for the faults to be detected might be paid different salaries). The equation for $delays^O$ is given by the following:

$$delays^O = \sum_{i=1}^{m} \left(\left(\sum_{k=1}^{TF_i^O} e_k^O \right) \times f_i \right)$$

In the prioritization case, cost savings are evaluated by comparing the application of a

prioritization technique that creates an order O'' relative to using a random order O'.

The cost of the random ordering is given by:

$$C^{O'} = delays^{O'}$$

The cost of a prioritization strategy is:

$$C^{O''} = delays^{O''} + Ca(T) + Cp(T)$$

The reduction strategy is more cost effective than retest-all if $C^{O'} < C^{O''}$. Substituting the above expressions results in:

$$delays^{O''} + Ca(T) + Cp(T) < delays^{O'}$$

This model can also be used to compare the cost-effectiveness of two prioritization techniques. Let O_1 and O_2 be the orders produced by two prioritization techniques. The corresponding regression testing costs are:

$$C_1 = Ca_1(T) + Cp_1(T) + delays^{O_1}$$

and

$$C_2 = Ca_2(T) + Cp_2(T) + delays^{O_2}$$

when techniques 1 and 2 are employed, respectively. Analysis costs and prioritization costs might differ. The technique with the lower C_k is the most cost beneficial.

2.7.6 APFD measure

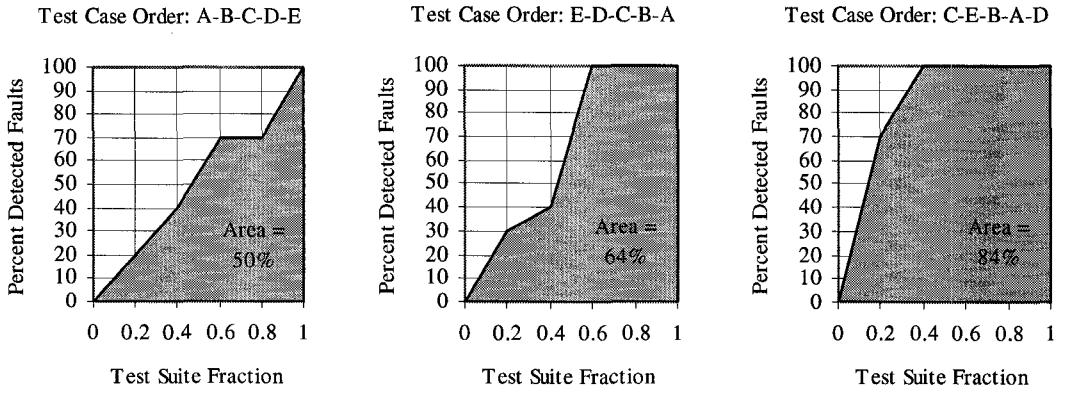
Rothermel et al. present a metric in [55] that can be used to measure the efficiency of prioritization techniques. This metric measures how quickly a test set detects faults. The metric is the weighted average of the percentage of faults detected (APFD) over the life of a test suite. The APFD value ranges from 0 to 100. The higher the APFD number is, the faster (i.e., the better) the rate of fault detection. The effectiveness of a variety of prioritization techniques for a given prioritized test suite can be compared by calculating

their APFD values.

Figure 2-4 illustrates the APFD measure. Figure 2-4a summarizes how the ten faults for an example program are detected by five test cases, A through E. Figure 2-4b shows the percentage of detected faults verses the fraction of the test suite run for the default test suite ordering, A-B-C-D-E. The area under the curve represents the APFD value over the life of the test suite. Figure 2-4c demonstrates the results for the test suite ordering, E-D-C-B-A; this is an ordering that detects the program faults earlier than the default ordering. Figure 2-4d demonstrates the optimal ordering, C-E-B-A-D, of the test cases; this sequence results in the earliest possible detection of the most faults.

Test	Fault									
	1	2	3	4	5	6	7	8	9	10
A	X				X					
B	X				X	X	X			
C	X	X	X	X	X	X	X			
D					X					
E								X	X	X

(a)



(b) (c) (d)

Figure 2-4 – Example illustrating the APFD measure.
(a) Test suite and faults exposed. (b) APFD for Prioritized Suite T1. (c) APFD for Prioritized Suite T2. (d) APFD for Prioritized Suite T3. (As presented by Rothermel et al. in [55].)

Elbaum et al. present the APFD equation in [18] as follows. Let T be a test suite with n test cases and let F represent a set of m faults revealed by T. Let TF_i be the first test case in the ordering T' of T that reveals fault i. The APFD for the test suite T' is given by the following equation:

$$APFD = \left[1 - \frac{TF_1 + TF_2 + \ldots + TF_m}{nm} + \frac{1}{2n}\right] \times 100$$

The previous APFD metric only applies in cases in which test costs and fault severity costs are uniform. Elbaum et al. create a "cost-cognizant" metric, $APFD_C$, in [16] that incorporates varying test case and fault severity costs. Test case cost might reflect a variety of measures such as time to execute, setup, and validate; or it could even reflect equipment or manpower costs. Similarly, fault costs could be measured in the time taken to locate and correct faults, or it might also reflect the cost of lost business, or the damage resulting from the faults.

The $APFD_C$ metric can be described as follows. Let T be a test suite containing n test cases with costs $t_1, t_2, \ldots, t_n$. Let F be a set of m faults revealed by T, and let $f_1, f_2, \ldots, f_m$ be the severities of those faults. Let TF_i be the first test case in an ordering T' of T that reveals fault i. The $APFD_C$ for the test suite T' is given by the following equation:

$$APFD_C = \left[\frac{\sum_{i=1}^{m}(f_i \times (\sum_{j=TF_i}^{n} t_j - \frac{1}{2}t_{TF_i}))}{\sum_{i=1}^{n} t_i \times \sum_{i=1}^{m} f_i}\right] \times 100$$

When test costs and fault severities are identical $APFD_C$ reduces to $APFD$.

2.8 Seeded Versus Naturally Occurring Faults

According to Do et al. in [11], typically naturally occurring faults are too costly to locate and cannot be found in sufficient numbers to support controlled experimentation; seeded faults, generated by hand-seeding or generated programmatically can be provided in large numbers allowing more data to be generated than otherwise possible.

Do et al. in [12] weigh the two approaches of seeding faults: mutation and hand seeding. They assert that the first approach would allow the generation of many faults but would not be representative of real faults; they also state that the second approach cannot cost-effectively produce large quantities of faults, but can generate more realistic faults than those generated by mutation.

It was generally believed that hand seeding provided more realistic faults than mutation faults but Andrews et al. in [1] suggest that mutation faults can be representative of real faults. Using mutation faults could potentially reduce the amount of effort required to configure any regression test experiments.

In [27], researchers at Siemens created faulty versions of seven base programs by manually seeding them with faults – usually by modifying a single line of code. Their goal was to introduce bugs that were as realistic as possible, based on their own programming experiences. Ten people performed the seeding, working mostly independently of one another.

Rothermel et al. in [47] followed a process of asking undergraduate and graduate students with at least 2 years of programming experience in C and unacquainted with the specifics of the study to become familiar with subject programs and to insert regression faults into the program versions. The fault seeders were instructed to insert faults that were as realistic as possible based on their experience with real programs, and that involved code deleted from, inserted into, or modified in the versions.

In [26], faults were selected such that they were "neither too easy nor too hard to detect." The logic was that faults that were too hard to detect would essentially result in fault detection rates of zero when regression testing techniques were applied. Conversely, it was thought that faults that were too easy to detect would be detectable by almost all resulting test sets. These conditions would rule out observing any meaningful differences between techniques.

Seeded faults must be reflective of regression faults. Rothermel et al. in [47] suggest using 10 seeded faults in each version of the program and they offer the following methodology for seeding faults:

1) Use the full regression test suite for the given version;

2) Seed faults (one at a time) that are as realistic as possible and that involve code deleted from, inserted into, or modified in the versions;

3) Execute test cases on the faulty program; and

4) If a fault is killed by at least one test case and by no more than 80% of the test cases, keep it otherwise discard it. (This is just one possible application of a rule to ensure that faults be "neither too easy nor too hard to detect.")

Observe that only faults that can be detected should be kept. Faults that are detected by more than 80% of test cases are deemed to be too easily detected and are discarded; the assumption being that such faults are easily detected by test engineers during their unit testing.

In [65], Wong et al. classify a variety of fault types: missing path, incorrect predicate, incorrect computation statement, missing computation statement, incorrect number of loop iterations, and missing clause in predicates. These faults can be mostly accomplished by changing a single line of code.

2.9 Summary of Previous Empirical Studies

There have been many empirical studies conducted in the field of software regression testing. Many of the programs used for performing empirical research have been used in multiple studies. Reuse of programs is advantageous, as the costs of creating software for evaluating regression testing techniques can be very high. To conduct a study on a given software program, it might be necessary to create test suites, faults, and even multiple versions in order to have the necessary infrastructure to evaluate the regression testing techniques.

Do et al. in [10] have performed an extensive review of a number of programs used in evaluating regression testing techniques. They have created an infrastructure to support the controlled experimentation of software testing and regression testing techniques, which serves to improve the validity of experimental results. Such an infrastructure provides a repository of software programs (in multiple versions if available), the tests associated with these programs, and the faults (seeded and/or real) associated with the version(s). The hope is that by making this repository available for use in a multitude of studies, the scientific validity of any results might be increased.

The repository is accessible online at [59]. Observing the nineteen programs currently available, only one of them, sed, a C program of 14 KLOC, is available in multiple versions and has real fault data available.

Appendix A summarizes some of the previous empirical studies that have been performed. Tables in Appendix A present the studies, the experimental techniques investigated, and provide descriptions of the subject programs that were used. Other tables in Appendix A summarize the software used in many of the studies, as well as how the test suites in the studies were designed and how fault information, if any, was obtained.

Of all the studies reviewed for this thesis very few make use of actual programs with multiple versions having real fault data. One such study [21] makes use of the player (subsystem of the internet game Empire) code for a study of regression test selection techniques; it should be noted that here the versions do not result from sequential modifications of the base program, "rather, each is a unique modified version of the base version." Another study [18] makes use of the QTB program for a study of test case prioritization; here sequential versions of software with real fault data are used. Other studies make use of programs with real sequential versions but only with seeded fault data. Given these observations, we can deduce that the use of real software versions and real fault data in experimentation is relatively rare and that this study will provide beneficial new data to add to the body of knowledge.

2.9.1 Studies conducted using procedural programming languages

Many empirical studies studying regression testing techniques have been performed on programs written in procedural languages (e.g., [3, 8, 14, 15, 16, 17, 18, 19, 21, 22, 24, 30, 31, 32, 37, 41, 46, 47, 48, 50, 53, 54, 55, 56, 65, and 66]). Table A-1 provides an overview of many of these empirical studies. Table A-2 contains descriptions of the programs used in these studies. Table A-3 further examines the techniques used to generate test cases and fault data for the programs used in some of the procedural studies.

2.9.2 Studies conducted using object-oriented programming languages

Fewer empirical studies relating to regression testing techniques have been performed on programs written in object-oriented languages (e.g., [11, 12, 25, 43, 52, 58, and 60]). Table A-4 provides an overview of some of the empirical studies conducted using OO languages. Table A-5 contains descriptions of the programs used in these studies. Table A-6 further examines the techniques used to generate test cases and fault data for the programs used in some of the OO studies.

2.10 Available Tools

Some of the previously reviewed regression testing techniques were implemented using prototype tools, or even simulated if appropriate tools were not available. Many of the experimental tools discussed are not readily available for our purposes as they have been implemented for use with UNIX. Other tools are available for use with Java. As we wish to experiment with regression testing on the Windows platform, we will limit our study to those tools that are available. The previously mentioned Microsoft prioritization tool, Echelon, is not commercially available.

Restricting ourselves to toolsets that are available for Windows, we identify four available packages that could be used as part of our regression testing study:

1. IPL Cantata++,
2. Telcordia Software Visualization and Analysis Toolsuite (*x*Suds),
3. IBM Rational Test RealTime, and
4. BullseyeCoverage.

Only *x*Suds offers specific regression testing functionality. All of these tools provide code coverage analysis that can be used to analyse software written in C and C++ on the Windows platform. A coverage tool allows us to evaluate the coverage of the experimental test suites that results when we execute our test cases. We can write additional test cases with this resulting information to increase the code coverage. Given

a collection of code coverage data associated with a set of test cases, one can implement a variety of the regression testing strategies – the difficulty in implementing such strategies would vary depending on the format of the data available and the complexity of the chosen regression testing approach. For example, it would be very easy to implement a total function coverage prioritization approach (with no feedback) given the function coverage statistics associated with each of the test cases in a test set.

The following sections serve to summarize the features of the previously listed software packages that might be of use to the proposed empirical study.

2.10.1 IPL Cantata++

The IPL webpage [28] sets out the following available features of this software:

1. **Code Coverage Analysis**
    a. Entry points and Call Returns,
    b. Statements and Basic Blocks,
    c. Decisions (branches),
    d. Conditions,
    e. MC/DC (DO-178B), and
    f. Exceptions.
2. **Customised Filtering.** Coverage can be viewed by test case (for test case optimisation).
3. **Context Coverage.** Cantata++ is aware of the context in which code has been covered. Inheritance and user defined contexts may be used (States, Threads).
4. **Relational Operator.** Coverage of equality / inequality operators for Equivalence Partitioning.

Of all the tools considered, Cantata++ is the only one that offers "context coverage". For example, it is possible to distinguish the resulting coverage of a base class that tests on a derived class have separately from coverage of the base class that might have been made through other means (e.g., by directly testing the base class).

2.10.2 Telcordia Software Visualization and Analysis Toolsuite (*x*Suds)

The *x*Suds manual [61] sets out the following functionality:

1. Support for block, decision, and data flow coverage testing;
2. Display of coverage summary information;
3. Display of uncovered source code;
4. Display of coverage overlap among test cases; and
5. Support for test set management and minimization.

The *x*Suds software allows us to perform modification-based regression test selection. *x*Suds additionally allows for minimization or prioritization techniques to be applied.

*x*Suds allows code coverage analysis to be performed for various code coverage criteria. Code coverage analysis is available for both C and C++ code. Table 2-9 summarizes the coverage analysis types as well as their language availability.

Table 2-9 – Code Coverage Analysis Types Available for the *x*Suds System.

Coverage Type	Description	Language Availability	
		C	C++
Function-entry	Ensures all functions are entered at least once.	Yes	Yes
Function-return	Ensures all explicit or implicit function returns or exits from a function are executed at least once.	Yes	Yes
Function-call	Indicates whether a call to a function has been made.	Yes	Yes
Block	Ensures that all blocks are executed at least once. A block is a code fragment not containing control-flow branching.	Yes	Yes
Basic Block	Ensures that all basic blocks are executed at least once. A basic block is a code fragments not containing control flow branching or function calls. The basic block option is provided as it is possible that a function call will not return. (For all the test cases of the NoiseGen software, this case never occurs.)	Yes	Yes

Decision	Ensures that each of the branches within a conditional statement evaluates to both true and false, at least once.	Yes	Yes
C-use	C-use (computational use) coverage. Ensures that if a variable is defined (assigned a value) and later used within a computation that is not part of a conditional expression, at least one path between this def-use pair is executed.	Yes	No
P-use	P-use (predicate variable use) coverage. Ensures that if a variable is defined and later used within a conditional expression, this def-use pair is executed at least once causing the surrounding decision to evaluate true, and once causing the decision to evaluate false.	Yes	No
All-uses	This is the sum of p-use and c-use coverage measures.	Yes	No

As the code in this study involves mostly C++, we cannot consider using C-use, P-use, or the All-uses coverage.

2.10.3 IBM Rational Test RealTime

The *IBM Rational Test RealTime – User Guide* [29] sets out the following available features of this software:

Table 2-10 – Code Coverage Analysis Types Available for IBM Rational Test RealTime

Coverage Type	Language Availability	
	C	C++
Block Coverage	Yes	Yes
Call Coverage	Yes	No
Condition Coverage	Yes	No
Function or Method Coverage	Yes	Yes
Templates	No	Yes

As the code in this study involves mostly C++ code, we can only consider Block Coverage and Function/Method Coverage.

2.10.3.1 Block coverage

The following block coverage types are offered:

1. Statement (or simple) Blocks

   Simple blocks are the function/method main blocks, blocks introduced by decision instructions:

   a. IF and ELSE;

b. FOR, WHILE and DO ... WHILE blocks;

c. non-empty blocks introduced by switch case or default statements;

d. true and false outcomes of ternary expressions (*<expr>* **?** *<expr>* **:** *<expr>*);

e. **TRY** blocks and any associated catch handler (C++ only); and

f. blocks following a potentially terminal statement.

2. Statement Blocks and Decisions (Implicit Blocks):

   Statement Blocks as described above.
   Implicit blocks are introduced by an **IF** statement without an **ELSE** or a **SWITCH** statement without a **DEFAULT**.

3. Statement Blocks, Decisions, and Loops (Logical Blocks):

   Statement Blocks and Decisions as described above.
   A typical **FOR** or **WHILE** loop can reach three different conditions:

   a. The statement block contained within the loop is executed zero times, therefore the output condition is *True* from the start;

   b. The statement block is executed exactly once, the output condition is *False*, then *True* the next time; and

   c. The statement block is executed at least twice. (The output condition is *False* at least twice, and becomes *True* at the end).

   In a **DO...WHILE** loop, because the output condition is tested after the block has been executed, two further branches are created:

   a. The statement block is executed exactly once. The output is condition *True* the first time; and

b. The statement block is executed at least twice. (The output condition is *False* at least once, then true at the end).

2.10.3.2 Function or method

Function or method coverage offers coverage measurements for the following:

1. Procedure Entries

   Inputs identify the C/C++ functions that are executed. One branch is defined per C/C++ function.

2. Procedure Entries and Exits (Returns and Terminal Statements)

   These include the standard output (if coverable), and all return instructions, exits, and other terminal instructions that are instrumented, as well as the input. At least two branches are defined per C/C++ function.

2.10.4 BullseyeCoverage

BullseyeCoverage [4] is a code coverage analyzer for C++ and C that offers "function coverage for quickly assessing overall coverage, and condition/decision coverage for detailed testing." They claim that their code coverage analysis is useful during unit testing, integration testing, and final release.

The following sections are BullseyeCoverage's overview of function coverage and condition/decision coverage [5].

2.10.4.1 Function coverage

Function coverage indicates whether each function was invoked or not. Function coverage allows the user to quickly assess what major areas of the target software are untested. To find defects quickly, first attain function coverage in all areas of the target software before delving into detailed testing of any one area.

2.10.4.2 Condition/decision coverage

Condition/decision coverage measures whether every control structure with every possible decision outcome as well as every possible condition outcome is exercised. A *decision* is the whole expression affecting the flow of control in a control structure. A decision may contain *conditions*, which are sub-expressions separated by logical-and (&&) and logical-or (||) operators. Condition/decision coverage basically measures whether all the logic of the control structures is tested.

Condition/decision coverage is the overall best coverage measure for detailed white-box code testing. Condition/decision coverage balances usability with thoroughness. Simpler coverage measures are blind to many obvious paths through the source code. For example, compared to line coverage, statement coverage, branch coverage, and condition coverage, condition/decision coverage is stronger and just as easy to use. Condition/decision coverage sees more test cases than these measures and therefore gives a better picture of what is not being tested. More thorough coverage measures require a significant increase in complexity combined with diminishing probability of exposing a bug.

3 EXPERIMENT DESCRIPTION

This section describes the steps taken in performing the experiments associated with this thesis. Section 3.1 presents a collection of research questions that serve to motivate this study. Section 3.2 provides a description of the subject software experimented upon. Section 3.3 describes the regression testing process undertaken for this study. Section 3.4 provides an overview of how the test sets were created to test the subject software. Section 3.5 explains how software faults were identified and counted. Section 3.6 explains why the software tool used to perform the experiments was selected. Section 3.7 proceeds to explain how this software tool is used to apply the various testing techniques to the subject software of the study.

3.1 Research Questions

Q1: How do the regression test selection techniques used in this study compare in terms of test set execution savings and fault detection effectiveness?

Q2: Do minimization techniques detect most of the faults detected by regression test selection techniques? Is the loss acceptable? And if it is, is the additional cost of minimization worth the observed test suite size reduction?

Q3: For the test reduction technique is there a trade-off in terms of test set execution savings and fault detection ability?

Q4: Can test prioritization improve the rate of fault detection of a regression test suite?

Q5: How do a variety of test case prioritization techniques compare to one another in terms of the rate of fault detection?

3.2 Program Under Study

This study seeks to apply regression testing techniques to an industrial software product. For confidentiality reasons, the actual name and version numbers of the software are not used; when describing the system, some of the terminology has been changed from that

of the original.

The software that is the subject of this study is used to control a hardware device. The software and hardware (hereafter referred to as the NoiseGen system) are bundled together in a rack-mount PC running the Windows operating system.

For this study, the exact operating system used is Microsoft Windows 2000 Professional with Service Pack 4; the computer is equipped with a Pentium III 750 MHz CPU and 256 MB of RAM.

The NoiseGen software in this study has typically undergone system level testing only during the regression testing phase. Therefore this is the only test phase will be addressed in this study. As White and Leung discuss in [64], the practice of performing regression testing only at the system level only is not recommended; they also recognize that unit and integration level testing are "often not applied consistently when changes are made."

3.2.1 NoiseGen overview

The NoiseGen system is used to simulate the impairments found on twisted pair copper wires. The NoiseGen system is typically used by the manufacturers of xDSL equipment to test the performance of their products. The NoiseGen software calculates noise profiles and then sends these profiles to an Arbitrary Waveform Generator (AWG) hardware card.

To generate a noise profile, a user first sets several parameters and then prompts the software to calculate and subsequently send this noise to one of the four AWG outputs. The parameters that a user may set include: a crosstalk noise file name, a crosstalk gain value, an RF ingress (RFI) noise file name, an RFI gain value, a sampling rate, a filter setting, the desired number of samples, etc. Once the user has entered all of the desired values, he or she issues a command to generate and then download the resulting noise profile to the hardware.

All versions of the NoiseGen software have been compiled using Microsoft Visual C++

6.0. They contain both C code and C++ code. Much of the C code has been generated from Matlab code (i.e., a Matlab compiler takes M program files and converts them into C code).

The NoiseGen software operates in two modes:

1. GUI mode. The user enters values on the GUI associated with this program.

2. Scripting mode. The user writes a script to manipulate the software settings. Commands are then sent to the NoiseGen via TCP/IP.

For the purposes of this study, the system will only be run in Scripting mode. The execution of the system test cases can therefore be easily automated.

3.2.2 The software components of NoiseGen

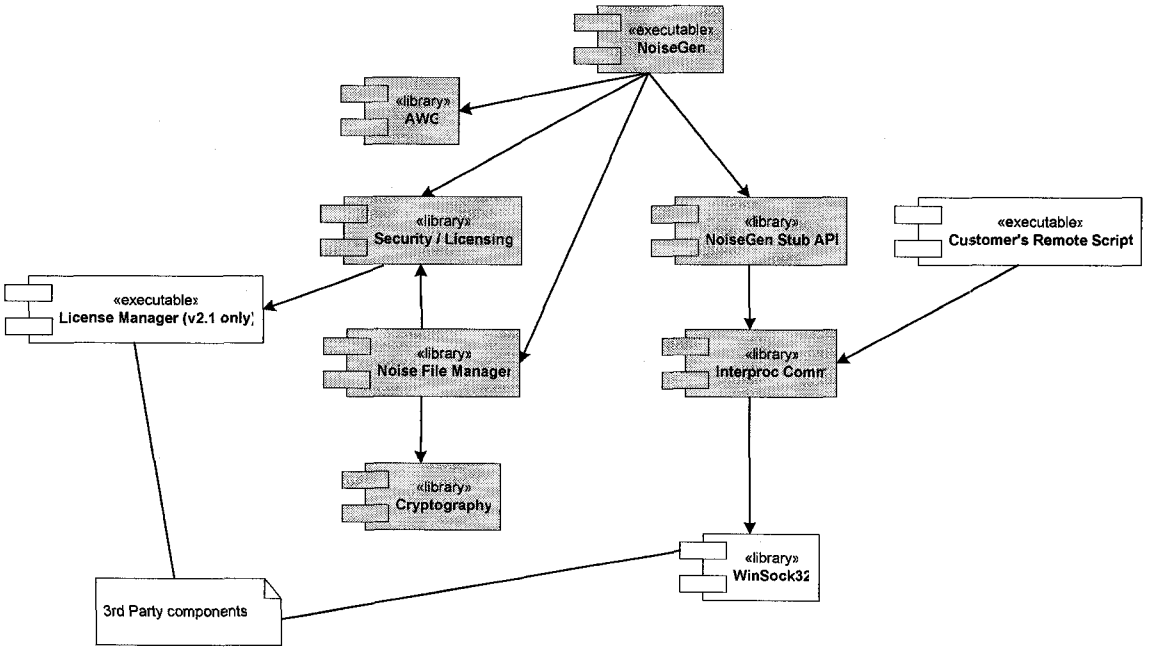


Figure 3-1 – Component view of the NoiseGen software.

The NoiseGen software is comprised of the following components, as illustrated in Figure 3-1:

1. NoiseGen Executable. This is the main executable, which provides a

computational engine for calculating and generating noise profiles as well as downloading noise profiles to the AWG hardware.

2. NoiseGen Stub API. Responsible for interpreting incoming remote scripting commands and then causing the NoiseGen system to act accordingly.

3. Interproc Comm. Provides support for TCP/IP communication.

4. AWG. Provides control for the AWG hardware.

5. Security / Licensing. Allows the licensing and activation of a variety of optional features. Optional features include: the ability to use certain noise input files, sold separately in packages (and protected by encryption); and other capabilities such as the ability to fine-adjust the AWG card output gain. Individual feature licensing is accomplished by issuing a specific license file for each system with encoded content to activate any purchased optional feature(s). In this study, initially, for version 1 a third-party license server is used in conjunction with the NoiseGen software to implement this feature. In later versions of the NoiseGen software, the use of the third-party license server software is dropped in favour of a complete in-house solution.

6. Noise File Manager. Its main task is to verify whether a user is licensed to use a protected noise file. If so, the file is decrypted and then is loaded into the NoiseGen software.

7. Cryptography. This library is used for the decryption of noise files.

Table 3-1 presents an overview of the NoiseGen version 1's components.

Table 3-1 – Code Overview of the NoiseGen Version 1 Software.

Component	Programming Language(s) Used	Lines of Code	Number of Classes	Number of Functions / Methods (listed per class)
NoiseGen Exe[1]	C	45411	0	Globals: 106
NoiseGen Stub API	C++	1035	1	Methods: 20 Globals: 5

Interproc Comm	C++	1170	8	Methods: 18, 6, 8, 4, 10, 7, 3, 5
AWG	C++	3904	6	Methods: 47, 36, 23, 29, 39, 6
Security / Licensing	C++	833	4	Methods: 6, 4, 12, 9 Globals: 18
Noise File Manager	C++	2109	8	Methods: 11, 5, 14, 10, 4, 15, 11, 11 Globals: 9
Cryptography	C++	288	3	Methods: 14, 14, 5
Total		54750	30	Methods and Globals: 544

[1] Many of the C files in this project have been generated via the Matlab C code generation utility. Most of the lines of code associated with this component are associated with these files (37058 lines of code).

3.2.3 The software versions of NoiseGen

Table 3-2 provides an overview of the evolution of the NoiseGen software. For each version, number of lines of code, and the nature of the changes made to the software is indicated.

Table 3-2 – Overview of the Evolution of the NoiseGen Software Package.

Version	Lines of Code	Nature of the Changes Made
1	54750	Initial version of the software considered for this study.
2	61615	Modifications made to noise file manager, security / licensing, NoiseGen Exe, and NoiseGen Stub.
3	59132	Modifications made to noise file manager, Interproc Comm, NoiseGen Exe, and NoiseGen Stub.
4	61090	Modifications made to noise file manager, NoiseGen Exe, and NoiseGen Stub.
5	67391	Modifications made to noise file manager, AWG, Interproc Comm, NoiseGen Exe, and NoiseGen Stub.
6	72925	Modifications made to noise file manager, AWG, NoiseGen Exe, and NoiseGen Stub.

Changes to the software were typically made to both the NoiseGen Stub and the NoiseGen Exe components in each version in order to accommodate new functionalities. Other modifications are made throughout the versions to correct problems. Major architectural changes were made in version 3; for example, Matlab code underwent a massive rewrite so that additional MFC GUI functionality could be added to the software.

As the software evolved, some of its scripting commands became obsolete. The policy adopted during the development of NoiseGen was that any obsolete commands should still be able to run in later versions of the software to maintain a backward compatibility

for any scripts that had been designed for the earlier versions; any obsolete commands are simply ignored by the command parser in the software. The behaviour of other scripting commands was changed from one version to the next such that a different output was expected due to specification changes.

Table 3-3 provides a breakdown of the scope of the modifications made to the software versions. Note that for the experiments, version 2 is considered as the base version of the NoiseGen software. As is discussed in section 3.5, we discard version 1 as the base version since no faults are detected in version 2 by its regression test suite. For our analyses, regression faults need to be present in order to provide useful data.

Table 3-3 offers code change information as provided by the *x*Suds tool (see 2.10.2). For each version, the total number of function entries, function returns, blocks, basic blocks, and decisions is listed. For each version, the number of changes made from the previous version is also listed for each of the coverage criteria. The changed code column is a tally of all code that has been modified or deleted in the previous version – added code is not included in the count.

Table 3-3 – NoiseGen Software Modification Information per Coverage Criterion.

Version	LOC	Function Entry		Function Return		Block		Basic Block		Decision	
		Total	Changed	Total	Changed	Total	Changed	Total	Changed	Total	Changed
2	61615	792	--	1066	--	15838	--	40853	--	8023	--
3	59132	1539	642	1988	692	15085	14292	29750	38998	9467	6697
4	61090	1562	605	2018	426	15662	2610	30657	3878	10039	2176
5	67391	1812	637	2365	454	17251	3851	32860	6525	11366	3121
6	72925	1848	565	2415	436	18762	4273	36766	7301	12134	3514

3.3 Application of the Regression Testing Process

This section explains in more detail how this experiment is executed. Mathur's regression testing process [39], as presented in section 2.2, will be followed. The steps presented in Figure 2-1, Figure 2-2, and Figure 2-3, of section 2.2 illustrate the experimental methodology followed.

Table 3-4 describes the number of test cases across the different software versions under study. Columns indicate: (A) the number of new test cases required to test new or modified functionality, from as many as 439 for version 1 to as few as 2 for version 6; (B) the total number of test cases available to test a version (these comprise the Full Test

Suite), from 439 for version 1 to 876 for version 6; (C) the number of test cases that fail for a given version, from as many as 158 for version 2 to as few as 18 for version 4; (D) the number of test cases from the previous version that are now obsolete, from as few as 0 for versions 2, 4, and 5, to as many as 36 for version 3; and (E) the total number of regression test cases available for a version (these comprise the Full Regression Test Suite), from 307 for version 2 to 849 for version 6. Note that column (E) is calculated by taking the total number of test cases (column B) from the previous version and subtracting the total number of test cases that fail (column C) in the previous version and then subtracting from this result the number of test cases from the previous version that are now obsolete (column D); the number of test cases that fail in the previous version cannot be considered as part of the regression test suite (see section 3.3.1 for further details).

Table 3-4 – Test Set Summary for Each of the NoiseGen Software Versions.

Version	(A) New Test Cases[1]	(B) Total Number of Test Cases in the Full Test Suite[2]	(C) Number of Test Cases that Fail	(D) Number of Test Cases from the Previous Version that are now Obsolete	(E) Total Number of Regression Test Cases in the Full Regression Test Suite[3]
1	439	439	132	--	--
2	102	541	158	0	307
3	228	733	134	36	347
4	39	772	18	0	599
5	103	875	25	0	754
6	2	876	34	1	849

[1] The new test cases include the replacement test cases for the obsolete test cases of the previous version. One new test case is created for each obsolete test case.
[2] (B) = Previous version's (B) + (A) – (D), for Version > 1; (B) = (A), for Version 1
[3] (E) = Previous version's (B) – Previous version's (C) – (D)

3.3.1 Test revalidation/selection/minimization/prioritization

Step 2 of Figure 2-1 depicts the revalidation step for a base version of the software.

Any failed tests (see Table 3-4, column C) are not considered in the regression test suite for the subsequent version of the software. This is a practical issue, as successful output files need to first be saved for each test case of the original release such that they may be used by the oracle when evaluating the results of the regression test suite for the subsequent release; without a successful output file from the previous software release,

the oracle cannot determine whether a test case has passed or failed. These failed tests are not permanently discarded; if the tests pass in the next version, they can then be included in the regression test suite going forward, provided that they have not become obsolete.

Figure 2-2 demonstrates the various regression test techniques that can be applied: coverage-focused minimization (step 3), total-coverage prioritization (step 4), additional-coverage prioritization (step 5), modification-based regression test selection (step 7), modification-focused minimization (step 8), total-coverage-using-modification-information prioritization (step 9), and additional-coverage-using-modification-information (step 10).

3.3.2 Test setup

The NoiseGen application is closed after each test is executed.

3.3.3 Test sequencing

As the NoiseGen application is initialized prior to running each test case, the test order does not affect test output. No dependencies exist between test cases. As no dependencies exist between test cases, test suite minimizations and prioritizations can be performed without the concern that the resulting test suite's test cases will not function as before.

The default sequence of the NoiseGen test cases is discussed in section 3.4.

3.3.4 Test execution

The process is fully automated. A resulting test text output file is saved for each of the test cases. Activity 11 of Figure 2-3 demonstrates this step.

Including the launch time of the NoiseGen application and the initialization of any necessary environment settings, a test script takes on average two minutes to execute. Note that this execution time is for the uninstrumented NoiseGen software.

3.3.5 Output comparison

Activity 12 of Figure 2-3 demonstrates this step. It should be noted that the oracle is not fully automated in this experiment. After the regression test suite is executed, Windiff, a graphical file comparison program, is used to compare the regression test output files against their expected values. The expected values are simply the saved output files resulting from the successful execution of the regression tests against the previous version(s) of the software.

The experimenter acts as the oracle, manually inspecting any differences found in the output files.

It should be noted in this experiment, that some of the test cases originally included in the regression test suite were found to be obsolete as a result of this step (some undocumented specification changes were found). In fact, for the NoiseGen software since all test scripts are backward compatible, by running the full test suite of the previous version, it is also possible using this output file comparison step to identify any obsolete test cases that the tester might not have identified earlier in the regression test validation step.

3.4 Test Set Creation

The category-partition strategy outlined by Ostrand and Balcer in [44] is initially used to create test cases that test the functionality of the system. Test specifications are made for each of the supported scripting commands. A test specification summarizes each command parameter that may be inputted as well as the environment settings that might influence the test outcome. Any constraints between input parameters and environment settings are also indicated. The NoiseGen's user manuals that summarize the use of the command set are employed to determine the commands and their associated input parameters. The user manuals are the main reference used to generate the test specifications; the experimenter's own knowledge of the product is also used in creating the specifications.

Given the formal test specifications, test frames are generated which consist of all

possible combinations of choices of input parameters and environment settings. Each test frame is then converted into test data (i.e., the actual parameter and environment values are assigned). The test data in turn is used to write a test script for input to the system. This test script, along with the expected system output, makes up a test case.

Code coverage information is then used to find any unexecuted areas of code and to guide the creation of additional test cases to generate additional coverage. Marick's advice [38] is then followed such that, rather than specifically targeting the missed code coverage condition, the tests written for the feature under test are reevaluated to address the testing design mistake that has led to the functionality being under-tested; focusing solely on code coverage in creating tests leads to test suites weak at finding faults of omission.

This study sought 100% coverage of coverable code. Recall from section 3.2.1 that the software has a GUI mode that is not tested.

Table 3-4 summarizes the number of test cases created for each version of the software. Columns indicate the number of new test cases written to accommodate a particular version and the total number of test cases available for a version.

3.4.1 Test set sequence

For the initial software version (NoiseGen version 1), the test cases are written for each command in turn. The default execution order of the tests is such that all the test cases for a given command are run before those of a successive command.

For each subsequent software version, additional test cases required to test any new functionality of the preexisting command set are added to the very end of the existing test set (i.e., the regression test suite). Following these test cases, additional test cases required to test the functionality of any newly added commands are added to the test set.

The next subsection specifically addresses the case where an existing test case has become obsolete.

3.4.2 Handling of obsolete test cases

As discussed in section 3.2.3, in certain cases, from one version to the next, NoiseGen commands were subject to specification changes. Given the same test input, a different output could now be expected. As a result, some of the test cases written to test command functionality were made obsolete. These obsolete test cases are not included in the regression test set for the new version.

Since NoiseGen scripting commands must be backward compatible, by extension all test scripts must be backward compatible; test inputs written for older software versions must always function on newer releases, even if the corresponding test outputs have changed. Therefore each of the obsolete test cases had to be replaced with a new test case (i.e., one containing the new expected output information).

Therefore, the test scripts associated with the obsolete test cases were rerun in the new version. The new test outputs resulting for each of these test scripts was recorded and saved to create the new test cases. These new test cases could then be added to the full regression test suite available for the next version (assuming no faults were found in the test output).

A test case created to replace an obsolete test case is placed in the same location of the test set sequence as the original obsolete test case.

3.5 Available Faults

This study makes use of real world faults as opposed to many of the previous studies that make use of seeded faults to simulate coding faults. Seeded faults are time consuming to produce but their advantage is that a large number of faults can be used to generate more experimental data, though with less external validity [10]. Real world faults are time consuming to locate and the number of faults that one will find is unknown at the outset. Of real concern is whether sufficient faults will be found to generate statistically significant data when the regression testing techniques are applied to the software.

As Munson et al. recognize in [40], the most widely used definitions of what faults are

makes it difficult to measure how many faults are present when a failure or failures occur: "There is no guarantee that two different individuals looking at the same set of failure reports and the same set of fault definitions will count the same number of underlying faults." They proceed to propose a solution that enumerates software faults based on the grammar of the software programming language used; the code modifications between the version that fails and the version in which the changes are made to correct the failure are required to create a fault count.

It would be difficult in this study to implement the Munson et al. approach to count faults. The NoiseGen code changes are generally poorly documented. Additionally, some of the software failures that were observed during testing were never corrected.

A study of the source code and other related in-house documentation pertaining to the NoiseGen software product did not yield much information as to the nature of any code corrections performed on the product. Windiff (a graphical file comparison program) was executed to observe the code changes made between versions in an attempt to locate any corrected faults. A few code corrections were observed between versions but it was not always obvious what faults were corrected by the changes made.

The following approach was ultimately taken to locate all the detectable code faults for each version:

1. For each of the versions under study, the full test suite available was executed.

2. The results of the test suite execution were analyzed manually and the test cases that failed were identified. For each of the commands, within each test script, the corresponding return value was analyzed to check that the expected value was outputted from the NoiseGen system.

3. The failed test cases were then further analyzed to determine what code fault or faults they detected. For each failed command, the observed failure was classified according to its behaviour. The failure was then attributed to a fault. Some test cases detected multiple faults and some faults were detected by multiple test cases.

This study relies on an ad hoc technique to identify and count individual code faults. Each time a failure is observed in the test output files, it was characterized accorded to the fault that occurred in the code to produce it. Accordingly, using this methodology, a fault may be caused by an error in a single statement of code or errors across many statements of code.

Note in step 1 that the Full Test Suite, not just the Full Regression Test Suite was executed. As discussed in section 3.3.1, this was done because it was necessary to determine if any new tests failed as these test cases could then not be included as a part of the regression test suite for the following version. Table 3-4 records the number of test cases that fail for each version. As a result of excluding these failed test cases from the full regression test suite, the number of regression faults that can be found in the next software version is potentially reduced.

The NoiseGen software initially underwent a much less vigorous testing regiment than that subjected to it by this study. It is not surprising that many of this study's test cases, written long after these software versions were released, result in failure.

Table 3-5 provides a detailed breakdown of the regression faults found for the software versions. A column presents the number of regression faults detected and the number of regression test cases that detect the faults for each version of NoiseGen. The number of faults detected is as low as zero for version 2, and as high as 30 (detected by 113 tests) for version 3. Another column presents the average number of fault detecting test cases per fault; it also expresses the average number of fault detecting test cases as a percentage of the full regression test suite size (see Table 3-4). Overall, the faults are detected by a small proportion of the full regression test suite: from as low as 0.37% of the full regression test suite for version 4 to as high as 1.41% of the full regression test suite for version 6. For each individual fault detected, the table also provides a column summarizing the corresponding number of fault detecting test cases.

Table 3-5 – Detailed Breakdown of the Number of Regression Faults per NoiseGen Version.

Version	Number of Regression Faults Detected and the Number of Regression Fault Detecting Test Cases[1]	Average Number of Fault Detecting Test Cases per Fault.[2] (Expressed as a percentage of the full regression test suite.)	Detailed Information per Fault	
			Fault Number	Number of Test Cases that Detect each Fault[1]
2	0	--	--	--
3	30 faults detected by 113 tests	4.27 (1.23)	1	14
			2	3
			3	9
			4	11
			5	3
			6	1
			7	3
			8	10
			9	2
			10	1
			11	3
			12	9
			13	12
			14	1
			15	2
			16	1
			17	2
			18	1
			19	1
			20	5
			21	1
			22	3
			23	13
			24	2
			25	2
			26	3
			27	4
			28	1
			29	3
			30	2
4	5 faults detected by 10 tests	2.2 (0.37)	1	1
			2	1
			3	1
			4	3
			5	5
5	1 fault detected by 10 tests	10 (1.33)	1	10
6	1 fault detected by 12 tests	12 (1.41)	1	12
All[3]	37 faults detected by 145 tests	4.35	--	--

[1] Some of the fault detecting test cases detect multiple faults.
[2] Multiple fault detecting test cases are counted multiple times, once for each fault occurrence.
[3] Across all versions, 37 faults are detected by 161 test cases, when multiple fault detecting test cases are counted multiple times, once for each fault they detect.

Obviously, many more faults might exist in the code than those that the test cases and oracle are able to detect. For this experiment, the recorded number of regression faults present in each version reflects only those faults detectable by the regression test suite.

Since a version having zero regression faults will not allow us to analyze the efficacy of regression testing techniques selected for the experiments, version 1 will be discarded and version 2 will be considered as the baseline for our experimental analyses.

3.6 Tool and Technique Selection

As indicated in the background study, the majority of the available regression testing tools are not implemented for the Windows platform. Microsoft has implemented a prioritization tool, but it is strictly for in-house use. Telcordia offers the *x*Suds tool that provides the following regression test functionality: modification-based regression test selection, test set minimization, and test set prioritization.

With the limited availability of tools directly supporting regression testing techniques, the use of code coverage tools to provide coverage data could be considered. An experimenter could then proceed to implement his or her own regression testing tools that make use of this coverage data. As mentioned in section 2.10, there are a number of tools available offering code coverage analyses. Cantata++ (see section 2.10.1) and BullseyeCoverage (see section 2.10.4) offer the most advanced coverage types for the C++ code that is the subject of this study. *x*Suds (see section 2.10.2) offers more advanced coverage techniques than does the IBM Rational Test RealTime (see section 2.10.3). *x*Suds offers decision coverage whereas IBM Rational Test RealTime does not offer the similar condition coverage option for C++ code.

Clearly *x*Suds offers a distinct advantage over the other tools that only offer code coverage analysis as those products do not directly support regression testing methodologies. The coverage information available from the other tools could potentially be taken and manipulated in such a way that they could be used in algorithms to implement regression testing techniques.

As it offers the most functionality for performing regression testing as an off-the-shelf tool, the *x*Suds tool was chosen for the experiments conducted in this study. It was used to measure code coverage and then to perform regression testing techniques. We limit ourselves to the coverage based techniques that are supported both in C and C++ languages. Table 2-9 provides a description for each of the coverage types selected for this study, namely: function-entry, function-return, block, basic block, and decision.

3.7 Applying Regression Testing Techniques to NoiseGen

3.7.1 Overview

Appendix B summarizes the *x*Suds command set used in this experiment. It also provides an overview of how the *x*Suds commands were used to implement the various regression testing techniques performed for this study.

Using *x*Suds, the NoiseGen software is instrumented such that it is possible to capture trace information corresponding to each test case for each of the versions. Using the coverage data associated with the trace information and, when applicable, the modification information recording the changes made between one software version and the next, the *x*Suds tool is used to implement a variety of regression testing techniques.

The following components (outlined in section 3.2.2) were excluded from the analysis:

1) Cryptography. The encryption and decryption computations are so laborious that adding the *x*Suds instrumentation results in unacceptably long execution times.

2) AWG. Attempting to run this component with xSuds instrumentation causes the NoiseGen software to fail on startup.

An examination of Table 3-1 reveals that these excluded modules combined account for only fraction of the overall source code, 4192 of 54750 lines of code (or 7.7% of all the system code), for version 1. In subsequent versions, these modules account for an even smaller fraction of the overall source code. As these modules account for such a small fraction of the overall code implementation and since none of the detectable faults were found in either of the excluded modules, we would expect that their exclusion will not affect the experimental outcomes significantly.

The following subsections describe the experiments performed for this experiment. Each subsection describes the regression test techniques applied to NoiseGen for the related experiment; these techniques are selected as *x*Suds readily supports their implementations. Table 3-6 provides an overview of which research question is addressed by which experiment.

Table 3-6 – Breakdown of Which Research Question is Addressed by Which Experiment.

	Research Question Addressed				
	Q1	Q2	Q3	Q4	Q5
Experiment A	X	X			
Experiment B		X	X		
Experiment C				X	X

As mentioned in the introduction to this thesis, this study only examined a subset of the existing regression testing techniques. Section 2.4 summarized the available techniques found in the literature for the regression test selection (section 2.4.2), test suite reduction (section 2.4.3), and prioritization (section 2.4.4) methodologies; in the subsections associated with each of the methodologies, the techniques supported by xSuds (and used in this study) were presented separately.

3.7.2 Experiment A – Study of regression test selection

The following regression test selection techniques are applied:

1. Retest-all (control technique),

2. Modification-based regression test selection, and

3. Modification-focused minimization.

Table 2-3 describes the *x*Suds implementation of the modification-based regression test selection and the modification-focused minimization techniques. Sections B.2.1 and B.2.2 of Appendix B outline how the tool suite commands are used to implement the modification-based regression test selection and the modification-focused minimization respectively.

Both modification-based regression test selection and modification-focused minimization are performed using the following coverage measures: function-entry, function-return, block, basic block, decision, and all the aforementioned techniques combined. Table 2-9 provides descriptions for each of these coverage types.

To perform a comparison of the effectiveness of these techniques, this study makes use of

the *percentage reduction in test suite size* metric as described in section 2.7.4.1 and the *percentage reduction in fault detection effectiveness* metric as described in section 2.7.4.2. These values are calculated for each coverage measure and for each of the versions that were regression tested.

An average *percentage reduction in test suite size* value is calculated for all the software versions combined for each coverage measure used. The technique merely sums all the test cases available for each of the versions to determine the total number of test cases under consideration; it similarly sums all the resulting selected test suite sizes from each of the versions to determine the total number of selected test cases.

An average *percentage reduction in fault detection effectiveness* value is similarly calculated by examining the total number of faults across the software versions and the total number of faults detected across the software versions.

This experiment addresses the following research questions: Q1 and Q2, by providing costs and fault detection data for the selection techniques studied.

3.7.3 Experiment B – Study of test suite reduction

For this experiment, test suite reduction is always performed on the full regression test suite available. This differs from the description of the test suite reduction technique given in 2.4.3 that seeks to "permanently remove redundant test cases in *T*"; if this approach were taken, then the minimization for a given version would only be performed on the minimized regression test suite available from the previous version.

Even though only a minimized regression test suite might be used to test a software version prior to its release, after its release occurs, we assume that there is sufficient time to run the entire regression test suite and then record the coverage data for that specific version. This data is then available to perform the minimization for the subsequent version.

The following test suite reduction techniques are applied:

1. No minimization (control technique), and

2. Coverage-focused minimization.

The coverage-focused minimization of Experiment B differs from the modification-focused minimization of Experiment A in that it considers the test case coverage of all the code. Modification-focused minimization considers only changed code for its minimization.

Table 2-5 describes the *x*Suds implementation of the coverage-focused minimization technique. Section B.2.3 of Appendix B outlines how the tool suite commands are used to implement the minimization.

The coverage-focused minimization is performed using the following coverage measures: function-entry, function-return, block, basic block, decision, and all the aforementioned techniques combined. Table 2-9 provides descriptions for each of these coverage types.

To perform a comparison of the effectiveness of these techniques, this study makes use of the *percentage reduction in test suite size* metric as described in section 2.7.4.1 and the *percentage reduction in fault detection effectiveness* metric as described in section 2.7.4.2. These values are calculated for each coverage measure and for each of the versions that were regression tested.

As explained in 3.7.2, an average *percentage reduction in test suite size* value and an average *percentage reduction in fault detection effectiveness* value is calculated for all the software versions combined for each coverage measure used.

This experiment addresses the following research questions: Q2 and Q3, by providing data that can be used to analyze the fault detection effectiveness and costs of the minimization techniques.

3.7.4 Experiment C – Study of prioritization techniques

For the prioritization techniques, version-specific prioritization is performed (see section 2.4.4.1). After each release, we assume that there is sufficient time to rerun and prioritize the full regression test suite for use with the next version.

The following prioritization techniques are applied:

1. No prioritization (control technique). This technique evaluates the desirability of using the untreated (default) test set sequence, as presented in section 3.4.

2. Test suite is ordered randomly (control technique).

3. Test suite is placed into optimal order such that software faults are executed the earliest possible. Recall this gives us a maximum bound for prioritization techniques.

4. Total-coverage approach.

5. Additional-coverage approach.

6. Total-coverage-using-modification-information approach.

7. Additional-coverage-using-modification-information approach.

Table 2-8 describes how each of the prioritization techniques is implemented. Sections B.2.4, B.2.5, B.2.6, and B.2.7 of Appendix B outline how the tool suite commands are used to implement the total-coverage, additional-coverage, total-coverage-using-modification-information, and additional-coverage-using-modification-information prioritizations respectively.

The prioritizations are performed using the following coverage measures: function-entry, function-return, block, basic block, and decision. Table 2-9 provides descriptions for each of these coverage types.

To perform a comparison of these techniques, this study makes use of the APFD metric as described in section 2.7.6. For each technique, this value is calculated for each coverage measure and for each of the versions that were regression tested.

An average APFD value is calculated for all the software versions combined for each technique. This is a weighted average that gives more significance to those test suites that detect more faults. Given that 37 faults are detected across the four software versions considered (30, 5, 1, and 1 faults in versions 3, 4, 5, and 6 respectively), then

$$APFD_{All} = \frac{30}{37} \times APFD_{ver.3} + \frac{5}{37} \times APFD_{ver.4} + \frac{1}{37} \times APFD_{ver.5} + \frac{1}{37} \times APFD_{ver.6}$$

This weighted average allows for generalizations to be made across all versions.

Recall that the optimal technique is for comparative purposes only. It allows us to compare the efficacy of the other techniques to the theoretically maximum possible APFD score. The optimal prioritization technique is performed manually by the experimenter.

This experiment addresses the following research questions: Q4 and Q5, by providing data to determine whether prioritization can significantly improve the speed at which faults are detected while executing a test suite and how do various prioritization techniques compare to each others.

3.7.5 Experimental repetitions

Random selection is a factor in some of the techniques described above. For example, if prioritization technique M4 (defined in Table 2-8) encounters multiple test cases covering the same number of function entries, they are ordered randomly. The experiments involving random selections should be repeated several times to observe and assess the resulting random variation in fault detection effectiveness.

It is observed when executing the *x*Suds tool that the results of both prioritization and minimization operations are influenced by the order in which the test case coverage data is listed in a trace file. This is because when there are tying coverage conditions shared by multiple test cases while performing a minimization or prioritization operation, *x*Suds relies on the order of the test cases in the file to resolve these ties – there is no random selection.

In order to collect statistically significant data that takes into account random selections, this study will use 100 randomly ordered trace files (each containing identical coverage data) for each software version under study for each of the modification-focused minimization (see section 3.7.2), coverage-focused minimization (see section 3.7.3), and

prioritization (see section 3.7.4) experiments. (Prioritization techniques M1 and M3 do not make use of the 100 random trace files as these techniques only consider one specific sequence of the test suite.)

4 RESULTS AND DISCUSSION

The following sections contain the results for each of the experiments examining the regression testing techniques.

4.1 Regression Test Selection—Experiment A

4.1.1 Retest-all

The following tables summarize the results of the retest-all control technique for each software version, and for all the software versions combined. Table 4-1 summarizes the resulting savings in absolute terms and using the *percentage reduction in test suite size* metric. Table 4-2 summarizes the resulting costs in absolute terms and using the *percentage reduction in fault detection effectiveness* metric.

Table 4-1 – Reduction in Test Suite Size for the Retest-all Technique.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)
3	0 (347)
4	0 (599)
5	0 (754)
6	0 (849)
All	0 (2549)

Table 4-2 – Reduction in Fault Detection Effectiveness for the Retest-all Technique.

Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)
3	0 (30)
4	0 (5)
5	0 (1)
6	0 (1)
All	0 (37)

The results of the retest-all control technique demonstrate that no savings in test set size results; accordingly, there is no reduction in fault detection effectiveness.

4.1.2 Modification-based regression test selection

The following tables summarize the results of the modification-based regression test selection for each of the code coverage types used, for each software version, and for all the software versions combined (to get average values across all versions). Table 4-3 summarizes the resulting savings in absolute terms and using the *percentage reduction in test suite size* metric. Table 4-4 summarizes the resulting costs in absolute terms and using the *percentage reduction in fault detection effectiveness* metric.

Table 4-3 – Reduction in Test Suite Size for the Modification-Based Regression Test Selection Strategy.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)					
	All Techniques Combined	Function Entry	Function Return	Block	Basic Block	Decision
3	0 (347)	0 (347)	0 (347)	0 (347)	0 (347)	0 (347)
4	0 (599)	1.2 (592)	0.2 (598)	0 (599)	0 (599)	0 (599)
5	0 (754)	0 (754)	0 (754)	0 (754)	0 (754)	0 (754)
6	1.6 (835)	1.6 (835)	1.8 (834)	1.6 (835)	1.6 (835)	1.6 (835)
All	0.5 (2535)	0.8 (2528)	0.6 (2533)	0.5 (2535)	0.5 (2535)	0.5 (2535)

Table 4-4 – Reduction in Fault Detection Effectiveness for the Modification-Based Regression Test Selection Strategy.

Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)					
	All Techniques Combined	Function Entry	Function Return	Block	Basic Block	Decision
3	0 (30)	0 (30)	0 (30)	0 (30)	0 (30)	0 (30)
4	0 (5)	0 (5)	0 (5)	0 (5)	0 (5)	0 (5)
5	0 (1)	0 (1)	0 (1)	0 (1)	0 (1)	0 (1)
6	0 (1)	0 (1)	0 (1)	0 (1)	0 (1)	0 (1)
All	0 (37)	0 (37)	0 (37)	0 (37)	0 (37)	0 (37)

In examining the results of the modification-based regression test selection, no test set size reduction is observed in most cases (Table 4-3). Where reductions are made, they

are minor. Because the tests are run at the system level, the likelihood that a test case covers modified code is high. By extension then, with most if not all test cases selected for each version, it is not unexpected that all the known faults are detected. The type of code coverage used has little impact on the results (Table 4-4).

4.1.3 Modification-focused minimization

Section C.1 of Appendix C contains the detailed results for the modification-focused minimization technique. For each of the code coverage types used, for each software version, and for all the software versions combined (to get average values across all versions), the median, minimum, maximum, mean, standard deviation, 2.5th percentile, and 97.5th percentile values for the *percentage reduction in test suite size* and the *percentage reduction in fault detection effectiveness* are presented in tables.

The following tables summarize the results of the modification-focused minimization. Table 4-5 summarizes the means and standard deviations of the resulting savings in absolute terms and using the *percentage reduction in test suite size* metric. Table 4-6 summarizes the means and standard deviations of the resulting costs in absolute terms and using the *percentage reduction in fault detection effectiveness* metric.

Table 4-5 – Reduction in Test Suite Size for the Modification-Focused Minimization Strategy.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)											
	All Techniques Combined		Function Entry		Function Return		Block		Basic Block		Decision	
	mean	stdev	mean	stdev	mean	stdev	mean	stdev	mean	stdev	mean	stdev
3	70.9 (101)	0 (0)	96.5 (12)	0 (0)	96.8 (11)	0 (0)	72.9 (94)	0 (0)	72.9 (94)	0 (0)	76.1 (83)	0 (0)
4	87.8 (73)	0 (0)	97.3 (16)	0 (0)	96.2 (23)	0 (0)	89.1 (65)	0 (0)	89.1 (65)	0 (0)	88.8 (67)	0 (0)
5	80.6 (146)	0 (0)	97.7 (17)	0 (0)	97.5 (19)	0 (0)	83.0 (128)	0 (0)	83.0 (128)	0 (0)	83.3 (126)	0 (0)
6	83.0 (144)	0 (0)	97.8 (19)	0 (0)	97.1 (25)	0 (0)	85.6 (122)	0 (0)	85.6 (122)	0 (0)	83.9 (137)	0 (0)
All	81.8 (464)	0 (0)	97.5 (64)	0 (0)	96.9 (78)	0 (0)	84.0 (409)	0 (0)	84.0 (409)	0 (0)	83.8 (413)	0 (0)

Table 4-6 – Reduction in Fault Detection Effectiveness for the Modification-Focused Minimization Strategy.

Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)											
	All Techniques Combined		Function Entry		Function Return		Block		Basic Block		Decision	
	mean	stdev	mean	stdev	mean	stdev	mean	stdev	mean	stdev	mean	stdev
3	19.4 (24.2)	1.9 (0.56)	100 (0)	0 (0)	98.4 (0.47)	1.7 (0.50)	22.7 (23.2)	1.9 (0.56)	22.7 (23.2)	1.9 (0.56)	20.4 (23.9)	2.3 (0.69)
4	82.0 (0.9)	12.9 (0.64)	100 (0)	0 (0)	91.2 (0.44)	10.0 (0.50)	85.8 (0.71)	9.1 (0.46)	85.8 (0.71)	9.1 (0.46)	82.0 (0.9)	12.9 (0.64)
5	0 (1)	0 (0)	100 (0)	0 (0)	100 (0)	0 (0)	0 (1)	0 (0)	0 (1)	0 (0)	0 (1)	0 (0)
6	0 (1)	0 (0)	100 (0)	0 (0)	100 (0)	0 (0)	0 (1)	0 (0)	0 (1)	0 (0)	0 (1)	0 (0)
All	26.8 (27.1)	2.5 (0.92)	100 (0)	0 (0)	97.5 (0.91)	2.1 (0.77)	30.0 (25.9)	2.2 (0.81)	30.0 (25.9)	2.2 (0.81)	27.6 (26.8)	2.7 (1.0)

The results of the modification-focused minimization vary by software version and by the coverage type used. Table 3-5 demonstrates that most faults are detected by few test cases and therefore reductions in the test suite size might have a significant impact on fault detection effectiveness. Verifying whether this is the case is the object of investigation here.

The use of the coarse coverage measures (i.e., function entry and function return) yields the highest reductions in test set size. However, the size reductions result in a total failure to detect any fault for all of the versions. From this experiment, we can conclude that using the coarse coverage measures is not an option.

The fine coverage measures (i.e., block, basic block, and decision coverage) offer substantial reductions in test set size savings across the versions (from 72.9% to 89.1%). Considering the average test size savings for all the versions combined, the test size savings is 83.8% or 84%, depending on the technique used.

For the fine coverage measures, the resulting reductions in fault detection effectiveness are varied across the versions (from 0% to 85.8%). For version 3, a 20.4% to 22.7% reduction in fault detection effectiveness is observed. For version 4, there is an 82.0% to 85.8% reduction in fault detection effectiveness. For versions 5 and 6, there is no reduction in fault detection effectiveness. Considering the average reduction in fault detection effectiveness for all the versions combined, the reduction in fault detection

effectiveness is 27.6% or 30.0%, depending on the technique used.

Examining version 3 should provide the most significant set of data as this version provides the greatest number of faults for analysis; there is less minimization in this case than for the other versions. This makes sense, as this version is subject to more code changes (refer to Table 3-3) than the others, hence there is more modified code to be covered thus requiring that more tests be selected. Another factor that must be considered is the test set size; we hypothesize that in general if there are fewer test cases available, there is less possibility for reductions as the number of coverage redundancies would increase as the test size increases (this observation was made in [54], though this study did not consider modification information). Version 4 experiences the greatest test size savings; considering this version experienced the fewest code changes it follows that relatively few tests need to be selected to cover these changes compared to the other versions. Versions 5 and 6 both experienced test size savings; the code changes involved for these versions are also of limited size.

In Table 4-6, the differences observed in the percentage reduction in fault detection effectiveness for the modification-focused minimization technique when applied to different versions might be due to the fact that some faults are more difficult to detect than others. Table 3-5 provides a summary of the number of test cases that detect the faults for all the versions of NoiseGen. The faults of version 4 are the most difficult to detect, followed by version 3. From Table 3-5 one can see that each fault of version 4 is revealed on average by 0.37% of the full regression test suite; for version 3, this average is 1.23%. The faults of versions 5 and 6 are relatively easier to detect, they are detected on average by 1.33% and 1.41% respectively of the available tests in the full regression test suite; though these versions, with only a single fault each, cannot on their own provide reliable data (i.e. luck may play a part in the selection of only a single fault). Overall one can hypothesize that faults that can be detected by many test cases have a higher probably of being detected by a minimized test set; there is more of a chance that a fault detecting test will be selected for the minimized test set.

The nature of the code changes might also influence whether the fault detecting test cases

are selected. One might hypothesize that the nature of the code modifications made to version 4 makes it more difficult to select the fault detecting test cases.

Using all the coverage techniques combined yields no improvement over using any of the finer techniques on their own.

Figure 4-1 presents the test set size savings across all the versions of all the modification-focused minimizations as a set of boxplots. Figure 4-2 presents the reduction in fault detection effectiveness across all the versions of all the modification-focused minimizations as a set of boxplots.

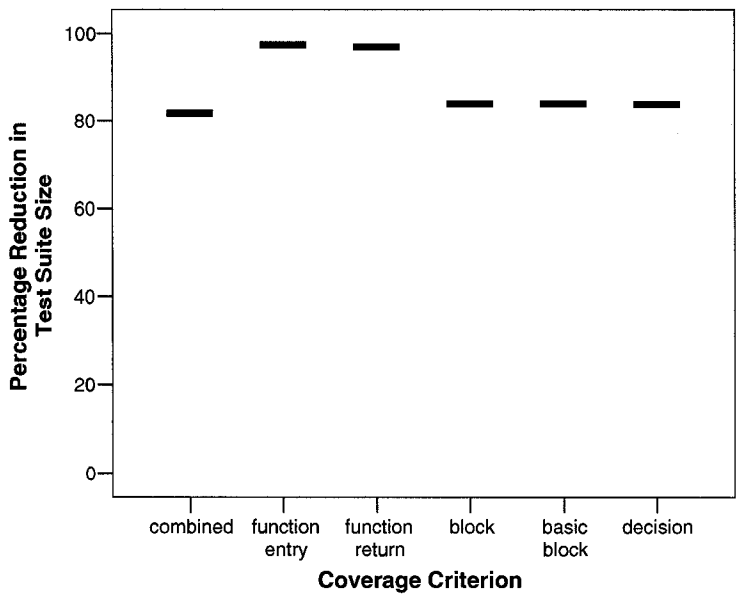


Figure 4-1 – Average test set size savings for the modification-focused minimizations across all the versions.

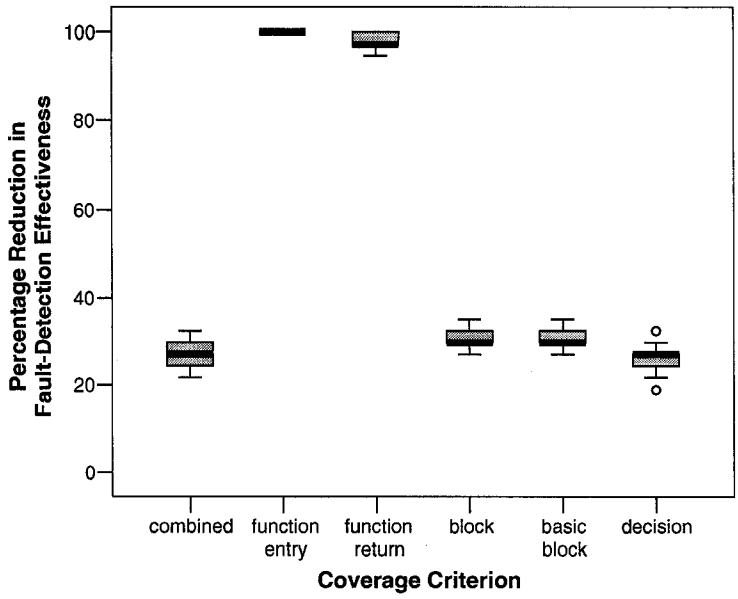


Figure 4-2 – Average reduction in fault detection effectiveness for the modification-focused minimizations across all the versions.

4.2 Test Suite Reduction—Experiment B

4.2.1 No minimization

The no minimization technique is identical to the retest-all technique of Experiment A. Table 4-1 summarizes the resulting savings in absolute terms and using the *percentage reduction in test suite size* metric. Table 4-2 summarizes the resulting costs in absolute terms and using the *percentage reduction in fault detection effectiveness* metric.

4.2.2 Coverage-focused minimization

Section C.2 of Appendix C contains the detailed results for the coverage-focused minimization technique. For each of the code coverage types used, for each software version, and for all the software versions combined (to get average values across all versions), the median, minimum, maximum, mean, standard deviation, 2.5^{th} percentile, and 97.5^{th} percentile values for the *percentage reduction in test suite size* and the *percentage reduction in fault detection effectiveness* are presented in tables.

The following tables summarize the results of the coverage-focused minimization. Table

4-7 summarizes the means and standard deviations of the resulting savings in absolute terms and using the *percentage reduction in test suite size* metric. Table 4-8 summarizes the means and standard deviations of the resulting costs in absolute terms and using the *percentage reduction in fault detection effectiveness* metric.

Table 4-7 – Reduction in Test Suite Size for the Coverage-Focused Minimization Strategy.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)											
	All Techniques Combined		Function Entry		Function Return		Block		Basic Block		Decision	
	mean	stdev	mean	stdev	mean	stdev	mean	stdev	mean	stdev	mean	stdev
3	66.3 (117)	0 (0)	96.5 (12)	0 (0)	95.4 (16)	0 (0)	70.3 (103)	0 (0)	70.3 (103)	0 (0)	72.0 (97)	0 (0)
4	80.3 (118)	0 (0.14)	97.3 (16)	0 (0)	91.7 (50)	0 (0)	82.1 (107)	0 (0.1)	82.1 (107)	0 (0.1)	81.8 (109)	0 (0.14)
5	77.9 (167)	0 (0)	97.6 (18)	0 (0)	93.1 (52)	0 (0)	81.0 (143)	0 (0)	81.0 (143)	0 (0)	79.3 (156)	0 (0)
6	76.7 (198)	0 (0)	97.5 (21)	0 (0)	93.4 (56)	0 (0)	80.1 (169)	0 (0)	80.1 (169)	0 (0)	78.2 (185)	0 (0)
All	76.5 (600)	0 (0.14)	97.4 (67)	0 (0)	93.2 (174)	0 (0)	79.5 (522)	0 (0.1)	79.5 (522)	0 (0.1)	78.5 (547)	0 (0.14)

Table 4-8 – Reduction in Fault Detection Effectiveness for the Coverage-Focused Minimization Strategy.

Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)											
	All Techniques Combined		Function Entry		Function Return		Block		Basic Block		Decision	
	mean	stdev	mean	stdev	mean	stdev	mean	stdev	mean	stdev	mean	stdev
3	19.3 (24.2)	2.0 (0.60)	100 (0)	0 (0)	98.3 (0.52)	1.7 (0.50)	22.7 (23.2)	1.9 (0.56)	22.7 (23.2)	1.9 (0.56)	20.3 (23.9)	2.4 (0.71)
4	77.8 (1.11)	14.5 (0.72)	100 (0)	0 (0)	100 (0)	0 (0)	77.8 (1.11)	14.5 (0.72)	77.8 (1.11)	14.5 (0.72)	65.8 (1.71)	9.1 (0.46)
5	0 (1)	0 (0)	100 (0)	0 (0)	100 (0)	0 (0)	0 (1)	0 (0)	0 (1)	0 (0)	0 (1)	0 (0)
6	0 (1)	0 (0)	100 (0)	0 (0)	100 (0)	0 (0)	0 (1)	0 (0)	0 (1)	0 (0)	0 (1)	0 (0)
All	26.1 (27.3)	2.7 (1.01)	100 (0)	0 (0)	98.6 (0.52)	1.4 (0.50)	28.9 (26.3)	2.6 (0.98)	28.9 (26.3)	2.6 (0.98)	25.4 (27.6)	2.4 (0.90)

Much like the modification-focused minimization, the results of the coverage-focused minimization also vary according to the coverage type used. Note that the resulting test set sizes are larger than those that resulted in the modification-focused minimization. This is to be expected as the modification-focused minimization technique functions by using the coverage data of the modified code only (see Table 2-3), while the coverage-

focused minimization deals with minimizing the test suite based on the coverage data associated with all of the code (see Table 2-5).

The use of the coarse coverage measures (function entry and function return) again yield the highest reductions in test set size but the size reductions result in a total failure to detect any fault for any of the versions.

The fine coverage measures (block, basic block, and decision coverage) again offer substantial reductions in test set size across the versions (from 70.3% to 82.1% of the original test set size). Considering all versions combined, the test size savings is 78.5% or 79.5%, depending on the technique used.

The resulting reductions in fault detection effectiveness (from 0% to 77.8%) are similar to the modification-focused minimization. Though for version 4, the fault detection effectiveness is higher. Considering all versions combined, the reduction in fault detection effectiveness is 25.4% or 28.9%, depending on the technique used.

Here, compared to the modification-focused minimization, one might hypothesize that the larger test suite size resulting from the coverage-focused minimization increases the probability that the test suite includes one of the fault detecting test cases.

Using all of coverage techniques combined yields no improvement over using any of the finer techniques on its own.

Figure 4-3 presents the test set size savings across all the versions of all the coverage-focused minimizations as a set of boxplots. Figure 4-4 presents the reduction in fault detection effectiveness across all the versions of all the coverage-focused minimizations as a set of boxplots.

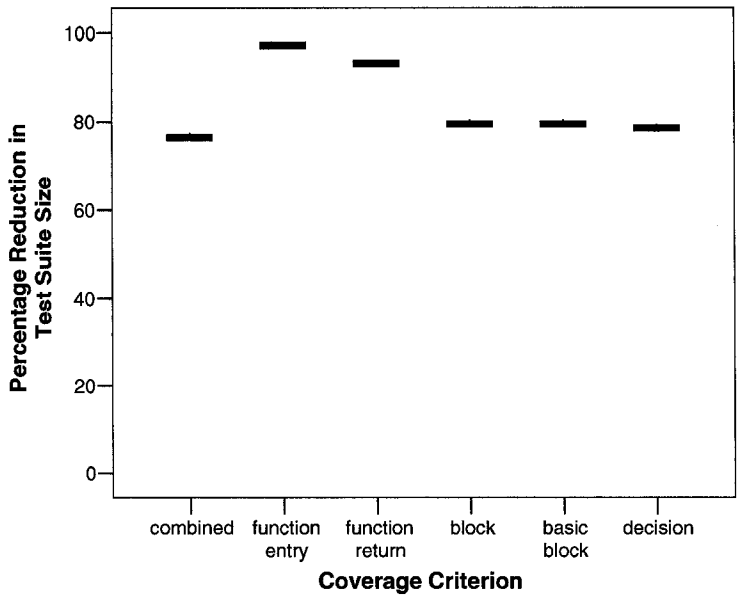


Figure 4-3 – Average test set size savings for the coverage-focused minimizations across all the versions.

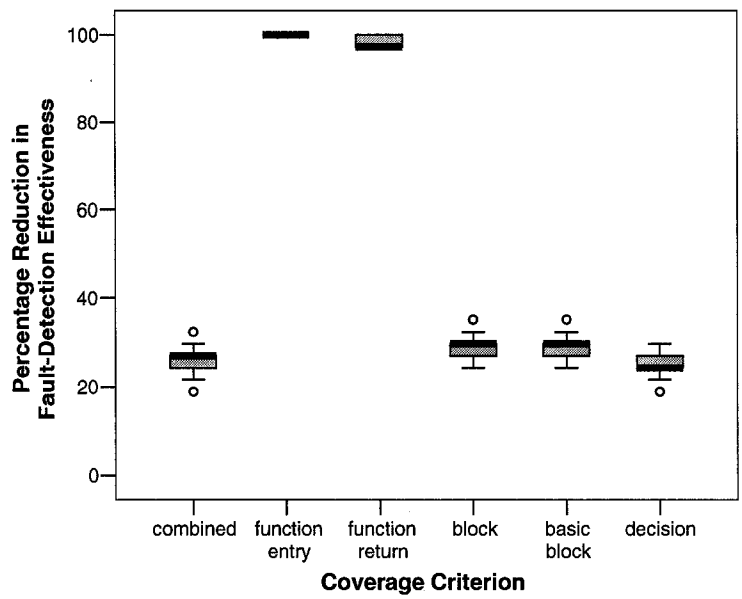


Figure 4-4 – Average reduction in fault detection effectiveness for the coverage-focused minimizations across all the versions.

4.3 Regression Test Prioritization—Experiment C

Section C.3 of Appendix C contains the detailed results for each of the prioritization techniques. The prioritization techniques are described in Table 2-8. For each of the techniques (except M1 and M3), for each of the software versions, and for the weighted average of the software versions, the median, minimum, maximum, mean, standard deviation, 2.5^{th} percentile, and 97.5^{th} percentile APFD values are presented in tables. Recall that techniques M1 and M3 examine only one sequence of the test suite, hence, only one APFD value is generated for each of the software versions; as discussed in section 3.7.5, all the other prioritization techniques are applied 100 times to provide statistically significant data to account for any random selections required when multiple test cases are tied (i.e., they offer the same code coverage).

The following tables summarize the results of the prioritization techniques for each of the software versions and for the weighted average (see section 3.7.4) of the software versions (denoted by 'All' in the tables). All results are reported using the APFD metric. Table 4-9 summarizes the APFD results of the control techniques (i.e., techniques M1, M2, and M3); in the case of M2, the APFD means and standard deviations are presented. Table 4-10 summarizes the APFD means and standard deviations of the total-coverage approach (i.e., techniques M4, M5, M6, M7, and M8). Table 4-11 summarizes the APFD means and standard deviations of the additional-coverage approach (i.e., techniques M9, M10, M11, M12, and M13). Table 4-12 summarizes the APFD means and standard deviations of the total-coverage-using-modification-information approach (i.e., techniques M14, M15, M16, M17, and M18). Table 4-13 summarizes the APFD means and standard deviations of the additional-coverage-using-modification-information approach (i.e., techniques M19, M20, M21, M22, and M23).

Table 4-9 – Prioritization Results for the Control Techniques.

Software Version	APFD Scores per Technique Used			
	M1	M2		M3
		mean	stdev	
3	56.9	70.9	4.0	97.1
4	23.4	61.8	13.6	99.7
5	92.2	91.3	7.8	99.9
6	95.8	92.2	7.2	99.9
All	54.4	70.8	3.6	97.6

The results of the no-prioritization (M1) and random ordering (M2) control techniques, presented in Table 4-9, will allow us to compare the results of the prioritization experiments (summarized in Table 4-10 to Table 4-13) to the results of some control techniques that require no special effort to implement. The results of the optimal ordering (M3) technique are also presented in Table 4-9. Technique M3 requires prior knowledge of the faults present in the code; its use is intended to give the upper bound for the APFD values for comparison.

Compared to all the other experimental techniques the no-prioritization technique (M1) results in the worst results. This is no surprise as test cases for NoiseGen are written sequentially in the order of the commands they test, as explained in section 3.4. Therefore any fault that is related to commands tested later in the test case sequence will take longer to reveal using no-prioritization (M1) than with the random (M2) or the other prioritization techniques (M4 to M23) that tend to redistribute the fault detecting test cases. The test case redistribution that results from using the random and other prioritization techniques increases the likelihood that faults will be revealed sooner. Across all versions, the $APFD_{All}$ score of M1 is 54.4, for M2 the mean $APFD_{All}$ score is 70.8, and for the other prioritization techniques (summarized in Table 4-10 to Table 4-13) the mean $APFD_{All}$ values range from 57.2 (for M14 and M15) to 77.2 (for M12).

Table 4-10 – Prioritization Results for the Total-Coverage Approach.

Software Version	APFD Scores per Technique Used									
	M4		M5		M6		M7		M8	
	mean	stdev	mean	stdev	mean	stdev	mean	stdev	mean	stdev
3	56.3	0.3	55.9	0.3	57.9	0.0	59.6	0.0	55.6	0.1
4	67.0	0.1	68.5	0.2	70.6	0.0	71.0	0.0	65.7	0.1
5	50.0	0.6	48.7	0.7	55.4	0.2	55.3	0.2	59.8	0.4
6	71.7	0.0	71.9	0.1	64.5	0.0	65.9	0.0	67.2	0.1
All	58.0	0.2	57.8	0.2	59.7	0.0	61.2	0.0	57.4	0.1

The results of the total-coverage techniques (M4 to M8), presented in Table 4-10, offer no improvement over the results of the no-prioritization (M1) and random (M2) control techniques, presented in Table 4-9. The use of the total-coverage techniques result in $APFD_{All}$ scores ranging from 57.8 to 61.2. This is only slightly higher than the resulting $APFD_{All}$ score of 54.4 from the no-prioritization (M1) approach and much worse than simply using the random approach, which provides a mean $APFD_{All}$ score of 70.8.

Observing the performance using the various coverage measures, there seems to be no clear advantage to using one measure over the other. For versions 4 and 5 the finer techniques (M6, M7, and M8) outperform the coarser ones (M4 and M5) though not by much. For version 6, the opposite is observed. The no-prioritization (M1) and random ordering (M2) techniques provide far superior results except in the case of version 4.

Table 4-11 – Prioritization Results for the Additional-Coverage Approach.

Software Version	APFD Scores per Technique Used									
	M9		M10		M11		M12		M13	
	mean	stdev	mean	stdev	mean	stdev	mean	stdev	mean	stdev
3	57.7	1.3	56.7	1.2	74.4	0.7	78.0	0.7	74.3	1.0
4	72.1	2.3	72.7	3.0	67.4	3.9	67.0	4.5	69.9	3.6
5	82.9	7.5	64.8	8.6	91.2	1.5	90.1	2.5	93.3	0.5
6	79.7	5.2	69.0	1.9	84.1	0.6	89.8	0.2	96.7	0.1
All	60.9	1.2	59.4	1.1	74.2	0.7	77.2	0.9	74.8	1.0

The additional-coverage techniques (M9 to M13) offer a significant improvement overall when compared to the total-coverage techniques (M4 to M8). This improvement is most pronounced for the finer grained coverage approaches (M11 to M13). The version 4 block coverage results (M11 and M12) are an exception to this. A comparison across the techniques yields the general observation that the finer grained techniques (M11 to M13) outperform the coarser grained ones (M9 and M10). This is especially true for version 3: the coarse grained techniques (M9 and M10) have APFD values in the mid-fifties; the fine grained ones (M11 to M13) in the seventies. The decision coverage (M13) type offers the best results for all but version 3, where the basic block coverage (M12) offers the best result. For versions 5 and 6, the decision coverage (M13) yields values approaching the optimal (M3) ones; one might guess that the fact that the faults of versions 5 and 6 are easier to detect than the other versions contributes to these high APFD scores, the nature of the code changes made for these versions might also be a factor in reaching the more favourable values. One should remember that versions 5 and 6 contain only one fault each so too much importance should not be attributed to the result; luck might also play a factor.

Across all versions, one observes that the coarser grained techniques do not fare better with the additional-coverage technique than they did with the total-coverage technique. For additional-coverage, the mean $APFD_{All}$ values are 60.9 (M9) and 59.4 (M10), from

Table 4-11, compared to values of 58.0 (M4) and 57.8 (M5), from Table 4-10, for the total-coverage.

For the finer grained techniques, there is a substantial improvement. For additional-coverage, the mean $APFD_{All}$ value ranges from 74.2 (M11) to 77.2 (M12), from Table 4-11, compared to values from 57.4 (M8) to 61.2 (M7), from Table 4-10, for the total-coverage.

Overall, using the finer grained additional-coverage techniques (M11 to M13) could be justified as they offer significant improvement over no-prioritization (M1) and random ordering (M2). Though there is still a lot of room for improvement when compared to the optimal (M3) case.

Table 4-12 – Prioritization Results for the Total-Coverage-Using-Modification-Information Approach.

Software Version	APFD Scores per Technique Used									
	M14		M15		M16		M17		M18	
	mean	stdev	mean	stdev	mean	stdev	mean	stdev	mean	stdev
3	55.6	0.3	55.6	0.3	58.3	0.0	59.7	0.0	55.7	0.1
4	65.0	0.1	65.6	0.2	65.0	0.0	66.9	0.0	64.2	0.1
5	52.9	0.3	46.2	1.0	61.1	0.3	63.9	0.1	64.6	0.1
6	70.8	0.5	76.9	0.7	72.4	0.1	74.6	0.0	68.0	0.1
All	57.2	0.3	57.2	0.2	59.6	0.0	61.2	0.0	57.4	0.1

The total-coverage-using-modification-information techniques (M14 to M18), presented in Table 4-12, do not offer any advantage over the total-coverage techniques (M4 to M8), presented in Table 4-10. A comparison of the $APFD_{All}$ values for each of the coverage criteria shows that they are nearly identical.

An improvement in the APFD score can be observed in versions 5 and 6 for most of the coverage approaches; though, as these versions only feature one fault each, too much importance should not be ascribed to these better results. A decrease in performance is observed for version 4 for all coverage approaches.

Version 3 does not seem to be influenced by the use of the total-coverage-using-modification-based strategy at all. This might be due to the nature of the major architectural changes made from version 2. Since so much code was either modified or

deleted (e.g., from Table 3-3 one observes that 6697 of 8023 decisions have been changed from version 2) the total-coverage-using-modification-information data might closely resemble the total-coverage data for each of the test cases. If this is the case, then one can hypothesize that the modification-based prioritization would effectively prioritize in a similar manner as the total-coverage approach.

As with the total-coverage approach, using the total-coverage-using-modification-information approach is not justified when compared to using the no-prioritization (M1) and the random ordering (M2) techniques.

Table 4-13 – Prioritization Results for the Additional-Coverage-Using-Modification-Information Approach.

Software Version	APFD Scores per Technique Used									
	M19		M20		M21		M22		M23	
	mean	stdev	mean	stdev	mean	stdev	mean	stdev	mean	stdev
3	57.6	1.4	60.9	1.2	75.0	0.7	78.4	0.7	76.4	0.8
4	72.5	2.4	63.3	3.4	60.4	2.8	62.2	2.9	67.2	3.3
5	67.2	7.7	67.9	5.4	93.0	0.7	91.8	1.4	96.3	0.2
6	70.8	2.2	60.4	3.9	96.2	4.9	92.7	0.2	99.6	0.0
All	60.2	1.2	61.4	1.0	74.1	0.7	77.0	0.7	76.4	0.8

The APFD scores of the additional-coverage-using-modification-information techniques (M19 to M23), presented in Table 4-13, offer a significant improvement over the APFD scores of the total-coverage-using-modification-information techniques (M14 to M18), presented in Table 4-12, for most cases for the finer grained techniques with the exception of version 4. The results for the coarser grained techniques are not consistent. In some cases an improvement is observed, as for version 4 using technique M19; the mean APFD is 72.5 for M19 compared to 65.0 for M14. In other cases a decrease in performance results as for version 6 using technique M20; the mean APFD is 60.4 for M20 compared to 76.9 for M15.

Considering the weighted average of all versions, one observes that the coarser grained techniques fare somewhat better with the additional-coverage-using-modification-information approach (M19 and M20, having $APFD_{All}$ values of 60.2 and 61.4 respectively) than they did with the total-coverage-using-modification-information approach (M14 and M15, both having identical $APFD_{All}$ values of 57.2). For the finer

grained techniques, there is a substantial improvement; for additional-coverage-using-modification-information techniques (M21 to M23), the mean $APFD_{All}$ ranges from 74.1 to 77.0 compared to values ranging from 57.4 to 61.2 for the total-coverage-using-modification-information techniques (M16 to M18).

The APFD scores of the additional-coverage-using-modification-information techniques (M19 to M23) can also be compared to the APFD scores of the additional-coverage techniques (M9 to M13), presented in Table 4-11. As was observed when comparing the APFD scores of the total-coverage-using-modification-information techniques (M14 to M18), presented in Table 4-12, to the total-coverage techniques (M4 to M8), presented in Table 4-10, version 3 does not seem to be influenced by the use of the modification-focused prioritization strategy. The same supposition can be made as before, that the large scope of the software changes causes the two strategies to behave similarly. For version 4, the performance decreases with the exception of the function-entry (M19) technique which remains essentially unchanged from before. For versions 5 and 6, the fine grained techniques using additional-coverage-using-modification-information (M21 to M23) produce superior APFD scores to the techniques using additional-coverage only (M11 to M13); for example, for version 5, technique M22 has an APFD score of 91.8 and technique M12 has an APFD score of 90.1. The coarser grained techniques using additional-coverage-using-modification-information (M19 and M20) produce inferior scores to the techniques using additional-coverage only (M9 to M10); for example, for version 6, technique M20 has an APFD score of 60.4 and technique M10 has an APFD score of 69.0.

Observing the weighted APFD average scores across all the versions, the performance of the additional-coverage-using-modification-information technique and the additional-coverage technique is similar for all coverage types. As with the finer grained additional-coverage techniques (M11 to M13), the finer grained additional-coverage-using-modification-information techniques (M21 to M23) are preferable to the no-prioritization (M1) and random ordering (M2) techniques. Though, the additional work involved with the additional-coverage-using-modification-information method is not justified; for additional-coverage-using-modification-information prioritization, the mean $APFD_{All}$

values range from 74.1 to 77.0 compared to almost identical values, from 74.2 to 77.2, for additional-coverage prioritization.

Figure 4-5 presents the results of all the prioritizations as a set of boxplots. The boxplots represent the $APFD_{All}$ values for each of the prioritization techniques studied. The prioritization techniques are described in Table 2-8.

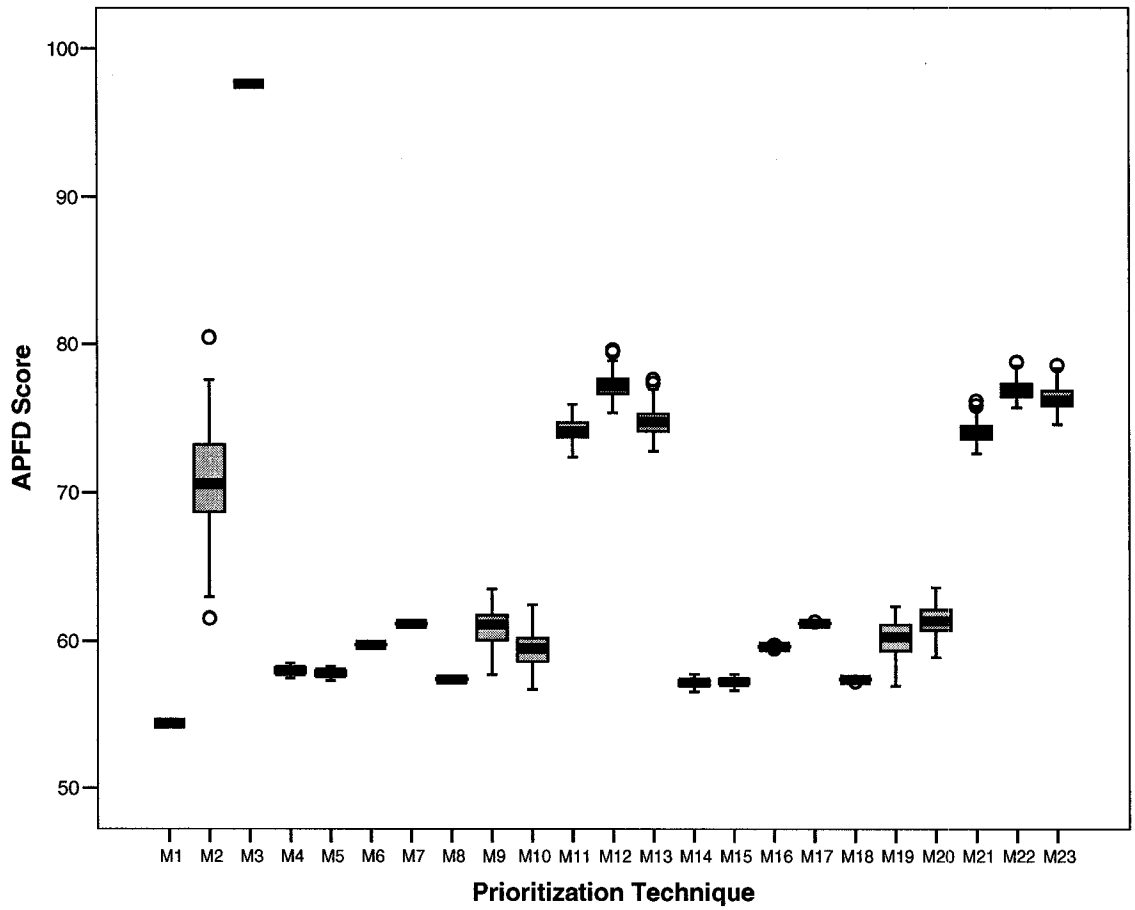


Figure 4-5 – Weighted average APFD scores for each of the prioritization techniques across all the versions.

Figure 4-5 indicates that the finer grained prioritization techniques using additional coverage (M11, M12, and M13) are the most effective; additional coverage using modification information techniques (M21, M22, and M23) perform similarly and do not improve the results. Using the additional coverage techniques outperforms the no prioritization (M1) approach as well as the random (M2) approach. The random

technique outperforms the no prioritization approach and all other heuristics (i.e., the total coverage approaches as well as the coarser grained additional coverage techniques). Even the best heuristics do not approach the optimal (M3) value.

4.4 Implications of Results

This study found that the use of the modification-based regression test selection strategy is not an effective strategy as it selects almost all of the test cases. This is not unexpected as quite a few modifications were made between the software versions under study; also, the nature of the modifications (e.g. the support of new commands to the NoiseGen system) required that core functionality be modified. Since the modification-based regression test selection of this experiment was performed at the system level, it is no surprise that little reduction in test size was made – most test cases covered the affected core functionality. This technique might yield a more favourable result if applied to a version with few modifications to non-core functionality.

The modification-focused minimization and the coverage-focused minimization using all the available code are able to produce major test suite size savings. Both minimization techniques, though, are likely to miss a lot of faults. The resulting loss in fault detection effectiveness makes these techniques unsuitable for regression testing any final software releases though they might provide an acceptable test solution for in-house testing during development. Preference should be given to the coverage-focused minimization and its comparatively larger test suite, should the time be available, due to its higher fault detection effectiveness.

The total-coverage prioritization techniques did not prove effective. The prioritization techniques using the additional-coverage approach and using the finer grained coverage criteria proved to be effective at increasing the speed of fault detection (APFD) when compared to no-prioritization and random ordering. Though, compared to the theoretical optimal, there is still significant room for improvement and it is clear that better prioritization solutions should still be sought out. The additional use of modification information did not improve APFD values.

Overall, the techniques using code modification information are not worthwhile. In the case of minimization, there is a slight increase in the loss of fault detection effectiveness when the modification information is used.

In the case of prioritization, overall the use of modification information is deemed to offer no discernable change in performance. However, examining individual version changes reveal a worsening performance in one case (version 4) and an improved performance in others (versions 5 and 6). Again we keep in mind that the versions 5 and 6 contain only one fault each; we cannot attach too much importance to these values as a result. A technique that could take into account the nature of the code modifications made in order to determine which technique is better suited would be beneficial.

For this experiment, the techniques using modification information were more costly, requiring more analysis work on the part of the tester (i.e., manually determining what files had been deleted since the last release and then generating the difference files for those files that had been changed since the last release); the required manual analysis could of course be automated. From the observed results, regardless of where modification information is used, whether in minimization or prioritization, it is not useful in general; therefore the use of modification information cannot be encouraged.

4.5 Comparing the Results to Other Studies

The following sections compare the results of this study to the results of previous studies. A selection of previous empirical studies as listed in Appendix A is used. Only the results of studies that used programs with 1000 LOC or greater are considered. Anything smaller is of lesser interest as they are probably not representative of real systems and difficult to compare with our case study.

4.5.1 Regression test selection

Table 4-14 contains a summary of the regression test selection results from this study and prior studies.

Table 4-14 – Comparing the NoiseGen Regression Test Selection Results to a Selection of Prior Studies.

Empirical Study	Techniques Used	Subject Programs	Results / Conclusions
Results of this study.			
An Empirical Study of the Regression Testing of an Industrial Software Product	- Modification-based regression test selection[a] - Modification-focused minimization[a] [a]using the xSuds function-entry, function-return, block, basic block, decision, and coverage criteria	NoiseGen	The modification-based regression test selection yields little savings in test size. For modification-focused minimization, finer grained coverage techniques yield a mean test size savings of 84% with a mean loss of fault detection of 30%. Coarser coverage techniques yield test size savings of 97% with a 100% loss of fault detection effectiveness.
Prior studies involving procedural programming languages.			
On Test Composition and cost-effective regression testing [46]	- Retest-all - Modified entity (TestTube)[a] - Modified non-core entity[a] - Modification-focused minimization[a] [a]using a function coverage criterion	bash, emp-server	The test suites were evaluated using tests of different sized input groupings. From a single input (grain), to multiple inputs (64 grains). The following test input granularities were used: 1, 2, 4, 8, 16, 32, and 64. For modified entity: For both bash and emp-server for all input granularities, all tests were selected with no loss in fault detection effectiveness. For both retest-all and modified entity, using coarser grained inputs reduced the test suite execution time. For modified non-core entity: For emp-server the reduction in fault detection effectiveness is less than 20%. For bash the reduction in fault detection effectiveness reaches a maximum of approximately 40%. The degree of the reduction in fault detection effectiveness varied with test input granularity and the methodology used to group the coarser test grains. The execution time savings was most apparent at the finer grained test level; for the coarser grained tests, the time savings decreased. For modification-focused minimization: For emp-server the reduction in fault detection effectiveness is between 10% and 30%. For bash the reduction in fault detection effectiveness is from 20% to over 60%. As with the modified non-core entity approach, the degree of the reduction in fault detection effectiveness varied with test input granularity and the methodology used to group the coarser test grains. The execution time savings was great compared to all other approaches.
Understanding the effects of changes on the cost-effectiveness of regression testing techniques [15]	- Retest-all - Modified non-core entity[a] - Modification-focused minimization[a] [a]using a function coverage criterion	bash, flex, grep, gzip	An analysis was done of the changes made to each software program to produce the "probability of execution of changed functions" metric to characterize the test suite; for the programs this value is: 13.8%, for bash; 49.3%, for grep; 28.1%, for flex; and 25.2%, for gzip. For the modified non-core entity technique, across all programs and versions the following approximate mean percentages of tests were selected from the test suites: for bash, 50%; for grep 100%; for flex, 60%; and for gzip, 40%. The following approximate mean percentage reductions in fault detection effectiveness were recorded: for bash, 10%; for grep, 0%; for flex, 30%; and for

			gzip, 10%. For the modified-focused minimization technique, across all programs and versions less than 8% of tests were selected from the test suites. The following approximate mean percentage reductions in fault detection effectiveness were recorded: for bash, 50%; for grep, 70%; for flex, 10%; and for gzip, 20%.
An Empirical Study of Regression Test Selection Techniques [21]	- Retest-all - Random[a] - DejaVu - Modification-focused minimization[b] [a]random test suites were created made up of 25%, 50%, and 75% of the tests of the full regression test suite [b]using a control-flow graph edge coverage criterion	space, player	For the DejaVu technique, across all versions the median reduced test suite size is roughly 20% for space, and 5% for player. As DejaVu is a safe technique, there is no reduction in fault detection effectiveness. For the modification-focused minimization technique, across all versions the median reduced test suite size is close to 0% for space, and is just slightly higher for player. The median percentage reduction in fault detection effectiveness is over 90% for space, and is roughly 0% for player. A reason for player's high level of fault detection effectiveness is that the versions of player contained multiple modifications causing a larger test suite to be selected (relative to space) for this technique, thereby increasing the odds of detecting faults. For the randomly selected test suites, the median percentage reduction in fault detection effectiveness for space approaches 100% when 25%, 50%, or 75% of the original test suite is selected; for player, the median percentage reduction in test fault detection effectiveness is roughly 85%, 95%, and 100%, for 25%, 50%, and 75% respectively, of the original test suite is selected.
Empirical Studies of a Safe Regression Test Selection Technique [50]	- DejaVu	player, Windows NT commercial program (dejavu simulated for player and commercial)	For player, the DejaVu technique reduces the test suite size on average over 95% over 5 versions. Considering both the analysis time and execution time combined, there is an average 87% time savings using this technique. For commercial, the test suite was evaluated using coarse grained tests (three inputs) or fine grained tests (388 inputs). (The fine grained tests comprised the coarse grained ones.) Over nine versions, under the 3-test interpretation, the number of tests was reduced by 48%; under the 388-test interpretation, the number of tests was reduced by 66%.
Prior studies involving object-oriented programming languages.			
Regression Test Selection for Java Software [25]	- DejaVOO	Siena, JEdit, JMeter, RegExp	The test reduction technique was shown to be able to reduce the size of a test suite by up to 98% to less than 10% for others. The selection was influenced by the number of changes and by the number of test cases that exercised the changed methods. The savings were greatest when the changes were minor and the changed methods were encountered by few test cases. An edge-level version of DejaVOO (manually conceptualized) was shown to be more efficient still then the method-level analysis provided by DejaVOO. Though the savings from running fewer test cases would have to justify the additional expense of the edge-level version.

Scaling Regression Testing to Large Software Systems [43]	- DejaVOO	Jaba, Daikon, JBoss	This paper studies a two phased approach of regression test selection. A high-level approach uses information about what classes and interfaces have changed to identify parts of the program that might be affected by changes between P and P'. The output of this phase is then subjected to further analysis and an edge-level test selection. An empirical study demonstrates that the approach scales well to large systems and that it produces large savings in regression testing time.
A Case Study of The Class Firewall Regression Test Selection Technique on a Large Scale Distributed Software System [58]	- class firewall technique - change based technique (test case is selected if it executes a changed class)	large distributed component J2EE system – banking software	Changed based approach outperforms the firewall regression test technique in reducing the test set size in this study. The cost of analyzing dependencies for the firewall technique may be too high to justify the use of the technique (i.e., vs. retest-all). The change based technique is the more cost effective technique in terms of the number of test cases selected for re-execution.
Regression test selection for C++ software [52]	- Control-flow-based technique	Tools.h++	The study demonstrates that the technique is able to reduce the number of tests required for regression testing. At a coarse granularity testing level, on average 62.5% of tests were selected. At a finer granularity level, on average 47.2% of test cases were selected.

A similar result is observed for the modification-based regression test selection as for the modified entity approach – a similar technique – observed by Rothermel et al. in [46]. The nature of the related software changes made is such that all test cases traversed changed code.

Using the modified non-core entity technique in other studies [45 and 14] yielded favourable test suite size reductions but also resulted in a loss of fault detection effectiveness. Both studies compared the modified non-core entity technique to the modification-focused minimization technique: the test size savings of the non-core entity technique was lower and the loss of the fault detection effectiveness was also lower. It might prove interesting to apply a modified non-core entity technique to the NoiseGen, but as a non-safe technique, one would guess that a reduction in fault detection effectiveness would occur.

Using the modification-focused minimization in studies [45, 14, and 20] yielded mostly similar results. Favourable reductions in test set size were made, but for the most part, unacceptable reductions in fault detection effectiveness resulted. Player in [20] did not suffer from a high loss of fault detection effectiveness as multiple modifications made to the software caused a large test suite to be selected.

Using DejaVu in [21, 50], DejaVOO in [25, 43], and the control-flow-based technique of [52] (these are all control-flow-based techniques) reveals that the technique was able to obtain some test set size savings in most cases; as safe techniques no loss in fault detection effectiveness occurred. This compares favorably to the results of this study where the test size could not be reduced without an unacceptable loss of fault detection effectiveness. However, the difference in results is most certainly due to the different software programs used in experimentation and can not be attributed to the technique used. For example, in study [25] great savings in test suite size were observed where few changes were made, "involving few methods, and methods encountered by only a few test cases."

4.5.2 Test suite reduction

Table 4-15 contains a summary of the test suite reduction results from this study and prior studies.

Table 4-15 – Comparing the NoiseGen Test Suite Reduction Results to a Selection of Prior Studies.

Empirical Study	Techniques Used	Subject Programs	Results / Conclusions
Results of this study.			
An Empirical Study of the Regression Testing of an Industrial Software Product	- coverage-focused minimization[a] [a]using the xSuds function-entry, function-return, block, basic block, decision, and coverage criteria	NoiseGen	For coverage-focused minimization, finer grained coverage techniques yield a mean test size savings of 80% with a mean loss of fault detection of 29%. Coarser coverage techniques yield test size savings of 97% with a 100% loss of fault detection effectiveness.
Prior studies involving procedural programming languages.			
On Test Composition and cost-effective regression testing [46]	- No reduction - GHS reduction[a] [a]using a function coverage criterion	bash, emp-server	The test suites were evaluated using tests of different sized input groupings. From a single input (grain), to multiple inputs (64 grains). The following test input granularities were used: 1, 2, 4, 8, 16, 32, and 64. For the GHS reduction: For both emp-server and bash the reduction in fault detection effectiveness is roughly between 10% and 20%. The degree of the reduction in fault detection effectiveness varied with test input granularity and the methodology used to group the coarser test grains. The execution time savings was greatest for the fine grained input granularity approaches.
Understanding the effects of changes on the cost-effectiveness of regression testing techniques [15]	- GHS reduction[a] [a]using a function coverage criterion	bash, flex, grep, gzip	An analysis was done of the changes made to each software program to produce the "probability of execution of changed functions" metric to characterize the test suite; for the programs this value is: 13.8%, for bash; 49.3%, for grep; 28.1%,

			for flex; and 25.2%, for gzip. For the GHS reduction technique, across all programs and versions less than 8% of tests were selected from the test suites (more tests were selected compared to the modification-focused minimization technique). The following approximate mean percentage reductions in fault detection effectiveness were recorded: for bash, 10%; for grep, 60%; for flex, 10%; and for gzip, 15%.
Test set minimization and fault detection effectiveness: A case study in a space application. [67]	- Random - *x*Suds minimization[a] [a]using block, decision, and all-uses coverage criteria respectively	space	Test sets were designed according to different criteria: by size, and by block coverage. The mean percentage savings was between 3% (for the smallest test set) and 85% (for the largest test set). The mean fault detection effectiveness reduction was between 2 and 7%. The xSuds minimizations were compared to random ones, the xSuds minimizations were found to be more effective than its random counterpart.
Empirical studies of test-suite reduction [54]	- Random minimization - GHS reduction[a] [a]using a control-flow graph edge coverage criterion	space	1000 sample test suites were created to obtain edge-coverage-adequate containing a random number of randomly selected test cases together with additional test cases necessary to achieve coverage. The test suites ranged in size from 159 to 4712 test cases. On average, the GHS reduction reduced test cases to a size of 124 test cases for all the sample test suites. This amounts to a reduction of between roughly 22% and 97%. The mean reduction in fault detection for was 8.9% for the GHS reduction. For the equivalent reduction for the test suites using random minimization, the mean reduction in fault detection was 18.1%.

Similarly to our findings, Rothermel et al. in [46] observed that the test suite reduction - in that study the GHS reduction (recall from section 2.4.3 that the technique is similar) - outperformed the modification-focused minimization with respect to fault detection effectiveness. As with the test suite reduction, using the GHS reduction also resulted in the creation of larger test suites when compared to the modification-focused minimization technique. In both this study and the Rothermel study, we might credit the resulting larger test suite sizes of the reduction techniques with their higher fault detection effectiveness when compared the fault detection effectiveness resulting from the smaller test suite sizes resulting from the modification-focused minimization technique.

The minimizations (GHS reduction in [15, 46, and 54] and xSuds minimization in [67]) varied with respect to the test suite size savings and reduction in fault detection

effectiveness. The observed size reductions vary greatly – from 3% to 97% of the original test suite. For some experiments test suites of varying sizes were used; for these experiments greater reductions were observed for the larger test suites due to an increase in test case coverage redundancies. In all cases, a reduction in test suite size resulted in at least some loss of fault detection effectiveness. The observed reduction in fault detection effectiveness was high in some studies (e.g., between 10% and 60% in studies [15 and 46]) and lower in others (e.g., less than 10% in study in [67]). From these values and from the results of this study one can conclude that the risk of a significant reduction in fault detection effectiveness is generally high using the minimization technique.

4.5.3 Prioritization

Table 4-16 contains a summary of the prioritization results from this study and prior studies.

Table 4-16 – Comparing the NoiseGen Prioritization Results to a Selection of Prior Studies.

Empirical Study	Techniques Used	Subject Programs	Results / Conclusions
Results of this study.			
An Empirical Study of the Regression Testing of an Industrial Software Product	(Summarized in Table 2-8) - Untreated - Random - Optimal - Function-entry-total - Function-return-total - Block-total - Basic-block-total - Decision-total - Function-entry-addtl - Function-return-addtl - Block-addtl - Basic-block-addtl - Decision-addtl - Function-entry-diff-total - Function-return-diff-total - Block-diff-total - Basic-block-diff-total - Decision-diff-total - Function-entry-diff-addtl - Function-return-diff-addtl - Block-diff-addtl - Basic-block-diff-addtl - Decision-diff-addtl	NoiseGen	The "total" and "diff-total" coverage approaches offer no advantage over the untreated and random orderings. The fine grained (i.e., block, basic block, and decision) "addtl" coverage approaches yield the best results; these results are favourable compared to the untreated and random approaches. The mean APFD value obtained for the basic-block-addtl technique, the most successful "addtl" approach, across all versions of NoiseGen is 77.3. Using the "diff-addtl" approaches did not improve the results of the "addtl" approaches. None of the results approached the optimal value.
Prior studies involving procedural programming languages.			
Understanding the effects of changes on the cost-effectiveness of regression testing techniques [15]	- Random - Optimal - Func-total - Func-addtl	bash, flex, grep, gzip	The func-addtl technique outperforms the func-total and random techniques for all four programs. For three of the four programs, the performance of func-addtl approaches that of the optimal approach.

			Only with bash does the func-total technique exceed the performance of the random technique. The func-addtl technique is recommended when there are many changes made across many functions and features. Func-total performed best when the changes occurred within the same feature, but these results were less impressive.
Test Case Prioritization: A Family of Empirical Studies [18]	- Random - Optimal - Func-total - Func-addtl - Func-FI-total - Func-FI-addtl - Func-diff-total - Func-diff-addtl	grep, flex, QTB	For both grep and flex, the func-addtl technique outperformed all others and approached the optimal results. For grep, with the exception of func-addtl, the random approach is as good as or better than the other approaches. For flex, both func-FI-addtl and func-diff-addtl techniques approach the performance of the func-addtl technique; they exceed the random result that in turn outperforms all other approaches. For QTB, it is the total techniques that are the better performers; though, these techniques fall short of the optimal approach. Func-total has a slight advantage over the func-FI-total and func-diff-total techniques; these techniques all outperform the random approach. None of the addtl approaches offer an advantage over the random approach. Of interest is that the use of feedback (i.e., the addtl as opposed to total techniques) did not always yield a better result. A possible explanation might be that faulty sections of code might be only partially covered by a test but then due to feedback, eventual fault revealing tests will be given a lower priority.
Prioritizing Test Cases for Regression Testing [55]	- Untreated - Random - Optimal - Stmt-total - Stmt-addtl - Branch-total - Branch-addtl - Stmt-FEP-total - Stmt-FEP-addtl	space	This study used both actual faults and mutants to calculate APFD values. For the study using actual faults, the stmt-FEP-addtl approach (with a mean APFD of 94.2) outperformed all others; though, it did not match the optimal value (with a mean APFD of 98.3). All other non-control techniques performed similarly (with mean APFD's of approximately 92); these in turn outperformed the random and untreated techniques (with mean APFD's of 83.4 and 83.3 respectively). For the study using mutants, the stmt-FEP-addtl approach (with a mean APFD of 92.1) outperformed all others; though it did not match the optimal value (with a mean APFD of 93.0). The branch-addtl and stmt-addtl approaches followed close behind (with mean APFD's of approximately 91). The 'total' approaches (with mean APFD's of approximately 85) did not perform well in comparison, though they still exceeded the untreated and random approaches (with mean APFD's of 82.5 and 81.1 respectively).
Selecting a Cost-Effective Test Case Prioritization Technique [19]	- Random - Optimal - Func-total - Func-addtl - Func-diff-total - Func-diff-addtl	bash emp-server flex grep gzip make sed xearth	Overall, the func-addtl technique outperforms all others (with the exception of the optimal). The 'addtl' techniques outperformed the 'total' ones; the func-diff-total technique outperformed the func-total technique.

Prior studies involving object-oriented programming languages.			
Empirical Studies of Test Case Prioritization in a JUnit Testing Environment [12]	- Untreated - Random - Optimal - Block-total - Block-addtl - Method-total - Method-addtl - Method-diff-total - Method-diff-addtl	Ant, XML-security, JMeter, JTopas	The associated test suite made use of test-classes that in turn were comprised of individual test-methods. Prioritization was performed at both the test-class and test-method levels. At the test-class level, prioritization techniques did not perform better compared to the untreated or random techniques. At the test-method level block-addtl and method-addtl far exceeded all others approaching the performance of the optimal.
A Controlled Experiment Assessing Test Case Prioritization Techniques via Mutation Faults [11]	- Untreated - Random - Optimal - Block-total - Block-addtl - Method-total - Method-addtl	Ant, XML-security, JMeter, JTopas	This experiment builds on the study of [12]. Mutation faults are now used to examine the prioritization techniques. The prioritization techniques outperformed the untreated and random techniques in all but a few cases. Compared to the results of [12], the results from this study showed less of a data spread for the results. This difference is due to the differing sizes of the mutation fault sets and the hand-seeded fault sets. Block-addtl and method-addtl are still the most successful techniques overall.

Overall, the "addtl" techniques outperform the "total" ones. Elbaum et al. in [14] observed that the func-addtl technique was preferable to the func-total one when multiple changes were made across many functions and features. As the NoiseGen was frequently subjected to multiple changes for each of the versions, this might explain the similarity in the observations – that the "addtl" techniques outperformed the "total" ones.

For the other studies, the techniques using more advanced code metrics (e.g., FI, FEP, or diff information) did not demonstrate any significant advantage, if any, over the simpler coverage-based approaches; this observation was also made for this study. Based upon these existing results, the use of the advanced code metrics cannot be recommended.

Overall though, none of the techniques presented get even near the optimal prioritization. This indicates that more research is still required as there is substantial room for improvement.

4.6 Answering the Research Questions

With respect to Q1 (i.e., how the regression test selection techniques compare in terms of test set execution savings and fault detection effectiveness), comparing the modification-based regression test selection strategy with the modification-focused minimization yields some major differences. The modification-based regression test selection strategy fails to achieve any meaningful test suite reductions (less than 1% savings averaged

across all the versions for each of the coverage criteria). Accordingly, for each of the coverage criteria, the reduction in fault detection effectiveness is 0% when compared to the "no selection" case. The modification-focused minimization yields significant reductions in test suite size (from 83.8% for the fine grained coverage criteria to 97.5% for the coarse grained coverage criteria averaged across all versions) and cost savings are achieved with respect to testing times. However, overall there is a high risk of a reduction in fault detection effectiveness (from 27.6% for the fine grained coverage criteria to 100% for the coarse grained coverage criteria averaged across all versions). Note that no reduction in fault detection effectiveness resulted in versions 5 and 6 for the fine grained coverage criteria. Any fault affecting the execution of the NoiseGen's command test set might have an adverse affect on the customers' own development schedules (recall that NoiseGen is used in the production of telecom equipment) and therefore any decrease in fault detection effectiveness is highly undesirable. Any problem that might affect the customer loyalty could be very costly not to mention the costs of having to reissue corrected software and of having to deal with individual customer complaints. Modification-focused minimization might prove an acceptable in-house testing technique during product development where sufficient time does not exist to fully retest this product on a daily basis. However, at least once before the final release of this product, the "no selection" (or retest-all) approach should be exercised to ensure that no faults are missed; the results of this experiment do not justify the use of the modification-focused minimization.

With respect to Q2 (i.e., whether minimization techniques detect most of the faults detected by regression test selection techniques, if the corresponding loss in fault detection effectiveness is acceptable, and if this loss is made up by the test suite size reduction), a similar observation can be made as above when comparing the modification-focused minimization strategy to the modification-based regression test selection strategy. The coverage-focused minimization is much like the modification-focused minimization strategy – the difference being that the modification-focused minimization considers the modified code only and the coverage-focused minimization considers all the code. Both minimizations result in similar outcomes. The modification-focused minimization produces slightly smaller test suites (with size savings from 83.8%

for the fine grained coverage criteria to 97.5% for the coarse grained coverage criteria averaged across all versions) compared to those test suites resulting from the coverage-focused minimization (with size savings from 78.5% for the fine grained coverage criteria to 97.4% for the coarse grained coverage criteria averaged across all versions); but the coverage-focused minimization has a slight advantage with respect to fault detection effectiveness (with an effectiveness reduction from 25.4% for the fine grained coverage criteria to 100% for the coarse grained coverage criteria averaged across all versions) compared to the modification-focused minimization (with an effectiveness reduction from 27.6% for the fine grained coverage criteria to 100% for the coarse grained coverage criteria averaged across all versions). While neither minimization approach would be an acceptable choice for the final regression test of the NoiseGen product, they might both be acceptable for in-house testing purposes during development. Due to the higher fault detection effectiveness of the coverage-focused minimization, this strategy is preferable though only if the increased time resulting from the larger test suite size can be accommodated in the test cycle.

With respect to Q3 (i.e., whether there is a trade-off between the test suite reduction and fault detection ability), the minimization technique is shown to dramatically reduce testing times. But the resulting reduction in fault detection effectiveness is significant and must be considered when testing the product with this technique. The cost of missing even a single regression fault might be deemed too high when measured against a negative customer reaction to a fault.

With respect to Q4 (i.e., whether test prioritization improves the rate of fault detection of a test suite), certain test prioritization techniques were shown to outperform the no-prioritization control technique (with an $APFD_{All}$ value of 54.4) and the random prioritization control techniques (with a mean $APFD_{All}$ value of 70.8). The fine grained techniques using additional coverage had consistently favourable results (with mean $APFD_{All}$ values ranging from 74.2 to 77.2) while those prioritizations using total coverage feedback fared poorly (with mean $APFD_{All}$ values ranging from 57.4 to 61.2).

With respect to Q5 (i.e., how a variety of test case prioritization techniques compare to

one another in terms of effects on rate of fault detection), as mentioned previously, overall the finer grained techniques outperformed the coarser grained ones. Overall, the additional coverage techniques outperform the total coverage techniques. The use of modification information does not improve the performance when code changes are extensive (as in version 3). When the code changes are more localized (as in versions 5 and 6) the use of modification information improves the performance (e.g., for version 5, with mean APFD value for M23 is 99.6 compared to M13 that has an APFD value of 96.7). When compared to the control techniques, the use of either of the additional-coverage or the additional-coverage-using-modification-information techniques seems justified. The decision coverage offers the best results for both approaches. Though the no-prioritization yields favourable results for version 5 and 6 (having APFD values of 92.2 and 95.8 respectively); it is essentially a lucky coincidence that fault detecting tests are run so early in the testing process. In some cases the random technique rivals some of the prioritization strategies. For example, in the case where large software changes were made and many faults existed (version 3) the random technique's performance (with an APFD of 70.9) rivals that of the more complicated strategies albeit with a higher standard deviation.

4.7 Threats to Validity

As discussed in the background section, many other studies make use of seeded faults. These studies seed a minimum number of faults per software version (e.g., 10 faults) and these seeded faults must be detected by a reasonable number of test cases (e.g., by at least one test and by no more than 80% of the tests). This is done to ensure a reasonably sized set of results from which to draw conclusions. In this study, NoiseGen versions 5 and 6 have only one fault each detected by few tests cases each. This does somewhat limit the conclusions that can be drawn from observing the associated experimental data. The fact that the faults of versions 5 and 6 are detected by few test cases, 10 and 12 respectively, does at least mean the faults are somewhat difficult to detect and subsequently, we can place some confidence in any technique that selects test cases that detects these faults.

Another issue related to this experiment is the fact that in the versions of software with multiple code faults, there is a possibility that a given fault detecting test case might

actually be able to detect more faults than is apparent. For example, a test case having executed a fault might fail; should this fault be corrected, re-executing the test case might now be able to detect another fault that was not previously detectable. If the faults were considered individually then the fault detection effectiveness values attained for the selections and minimizations and the APFD scores achieved for the prioritizations might be somewhat different. (This would of course require that each of the faults be corrected in the software.)

This experiment makes the assumption that all test cases are equal in cost. This is generally true in that each test script takes on average two minutes to execute and no test case requires any more effort than any other to execute and evaluate. There are slight variations in execution time (measurable in seconds) that could be taken into account to produce a more accurate result.

This experiment also makes the assumption that the costs associated with missing faults are equal. It is possible that some faults might cause greater problems for customers and may therefore have a higher associated cost.

This experiment only evaluated the effectiveness of the regression testing methodologies by executing regression tests against actual release versions. Had the regression tests been executed during actual development from one day to the next, different observations might have been made. For example, one can hypothesize, that the modification-based regression test selection would have successfully reduced the regression test set on a day-to-day basis since the code modifications would be isolated and smaller in scope and more conducive to such a selection technique.

Although experimenting on a single software package enables us to generate interesting data for discussion, any claims that can be made pertaining to the resulting data is limited. Ideally, a family of experiments conducted on different software programs of multiple versions should be conducted in order to make generalizations with respect to the results in [10].

4.8 Considering Only the Experimental Results of NoiseGen Versions 3 and 4

As mentioned in section 4.7, versions 5 and 6 of NoiseGen contain only one fault each and there is a concern that the inclusion of this data might threaten the validity of the observations made. Accordingly, Appendix D details the experimental results considering only versions 3 and 4 of the NoiseGen software. Averages using only these results are calculated.

The exclusion of versions 5 and 6 from the calculated averages does not result in any major differences. The resulting tables and figures in Appendix D closely resemble those that were generated using the experimental data from all of the NoiseGen versions.

5 CONCLUSION

5.1 Summary and Conclusions

This study contributes more empirical data to the existing body of regression test technique studies. This case study is distinct from many other studies in that it is one of only a few to apply regression testing techniques to a non-trivial real-world software program released in multiple versions (refer to section 2.9). A further advantage over other studies is its use of actual fault data. As in many other studies, the test suites used in this thesis were designed with the intent of conducting a variety of regression testing experiments; though, one can argue that the resulting test suites accurately reflect a real-world implementation as the techniques used to create the experimental test suites are the same as would have been used had the regression test suites been implemented during the actual development of the product. Examining two other studies that were observed to involve experimentation on more realistic real-world software programs, study [21] examined selection techniques and study [18] examined prioritization techniques only; this study applied selection techniques, reduction techniques, and prioritization techniques to a real-world software program.

For this particular study, modification-based regression test selection was shown to be ineffective due to the extensive nature of the code changes between each of the versions. Both modification-focused minimization and coverage-focused minimization were shown to yield significant savings in test suite size though with a corresponding loss in fault detection effectiveness that is deemed to be unacceptable for a final product. These techniques can however be considered during development for the regression testing of intermediary versions that are not delivered to customers. Only the finer grained additional-coverage prioritization techniques were shown to yield improvements in the regression testing process; the use of modification information was not beneficial to this process. However, there is still substantial room for improvement regarding the tested prioritization techniques and more research is required in this direction.

5.2 Future Work

Further analysis could be performed on the NoiseGen software to determine the nature of the code changes made and to determine whether the characteristics of these changes have an impact on the effectiveness of the regression test techniques. Previous studies have suggested that the types of code changes made can affect the efficacy of any regression test techniques.

A study comparing the use of varying test grain sizes on the cost and effectiveness of regression testing techniques could be performed. The test cases of this study were each initialized and run individually. By launching the NoiseGen software and running multiple test cases, the execution time could be greatly reduced; the corresponding effect on cost and effectiveness is of great interest.

As only a subset of the available regression testing techniques has been investigated, follow-up studies would ideally include additional techniques for analysis. Also, the creation of a more powerful oracle, able to evaluate the internal state of the software, could potentially reveal more faults and increase the data available in future studies.

6 REFERENCES

1. J.H. Andrews, L.C. Briand, and Y. Labiche. "Is mutation an appropriate tool for testing experiments?" in *Proc. of the 27th International Conference on Software Engineering (ICSE) 2005*, pp. 402-411, May 2005.

2. T. Ball. "On the limit of control flow analysis for regression test selection," in *Proc. of ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 134-142, Mar. 1998.

3. J. Bible, G. Rothermel, and D. Rosenblum. "A comparative study of coarse- and fine-grained safe regression test selection techniques," *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 2, pp. 149-183, Apr. 2001.

4. Bullseye Testing Technology. BullseyeCoverage Website. http://www.bullseye.com. Accessed: Nov. 13, 2006.

5. Bullseye Testing Technology. "BullseyeCoverage Measurement Technique," http://www.bullseye.com/measurementTechnique.html. Accessed: Nov. 13, 2006.

6. M.-H. Chen and H. M. Kao. "Testing object-oriented programs – An integrated approach," in *Proc. of the 10th International Symposium on Software Reliability Engineering*, pp. 73-82, Nov. 1999.

7. W. Chen, R. H. Untch, G. Rothermel, S. Elbaum, and J. von Ronne. "Can fault-exposure-potential estimates improve the fault detection abilities of test suites?," *Journal of Software Testing, Verification, and Reliability*, vol. 12, no. 4, pp. 197-218, Dec. 2002.

8. Y. F. Chen, D. S. Rosenblum, and K. P. Vo. "TestTube: A system for selective regression testing," in *Proc. of the 16th International Conference on Software Engineering*, pp. 211-220, May 1994.

9. M. E. Delamaro and J. C. Maldonado. "Proteum – A tool for the assessment of test adequacy for C programs," in *Proc. of the Conference on Performability in Computing Systems (PCS 96)*, pp. 79-95, Jul. 1996.

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

11. H. Do and G. Rothermel. "A controlled experiment assessing test case prioritization techniques via mutation faults," in *Proc. of the 21st IEEE International Conference on Software Maintenance*, pp. 411-420, Sep. 2005.

12. H. Do, G. Rothermel, and A. Kinneer. "Empirical studies of test case prioritization in a JUnit testing environment," in *Proc. of the 15th International Symposium on Software Reliability Engineering*, pp. 113-124, Nov. 2004.

13. S. Elbaum, D. Gable, and G. Rothermel. "The impact of software evolution on code coverage information," in *Proc. of the IEEE International Conference on Software Maintenance*, pp. 170-179, Nov. 2001.

14. S. Elbaum, D. Gable, and G. Rothermel. "Understanding and measuring the sources of variation in the prioritization of regression test suites," in *Proc. of the 7th International Software Metrics Symposium*, pp. 169-179, Apr. 2001.

15. S. Elbaum, P. Kallakuri, A. G. Malishevsky, G. Rothermel, and S. Kanduri. "Understanding the effects of changes on the cost-effectiveness of regression testing techniques," *Journal of Software Testing, Verification, and Reliability*, vol. 13, no. 2, pp. 65-83, Jun. 2003.

16. S. Elbaum, A. Malishevsky, and G. Rothermel. "Incorporating varying test costs and fault severities into test case prioritization," in *Proc. of the 23rd International Conference on Software Engineering*, pp. 329-338, May 2001.

17. S. Elbaum, A. G. Malishevsky, and G. Rothermel. "Prioritizing test cases for regression testing," in *Proc. of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 102-112, Aug. 2000.

18. S. Elbaum, A. Malishevsky, and G. Rothermel. "Test case prioritization: A family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159-182, Feb. 2002.

19. S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky. "Selecting a cost-effective test case prioritization technique," *Software Quality Journal*, vol. 12, no. 3, pp. 185-210, Sep. 2004.

20. K.F. Fischer, F. Raji, and A. Chruscicki. "A methodology for retesting modified software," in *Proc. of the National Telecommunications Conference*, B-6-3, pp. 1-6, Nov. 1981.

21. T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. "An empirical study of regression test selection techniques," *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 2, pp. 184-208, Apr. 2001.

22. T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. "An empirical study of regression test selection techniques," in *Proc. of the 20th International Conference on Software Engineering*, pp. 188-197, Apr. 1998.

23. M. J. Harrold. "Testing evolving software," *Journal of Systems and Software*, vol. 47, nos. 2-3, pp. 173-181, Jul. 1999.

24. M. J. Harrold, R. Gupta, and M. L. Soffa. "A methodology for controlling the size of a test suite," *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 3, pp. 270-285, Jul. 1993.

25. M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. "Regression test selection for Java software," in *Proc. of the 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pp. 312-326, Oct. 2001.

26. M. J. Harrold, D. Rosenblum, G. Rothermel, and E. Weyuker. "Empirical studies of a prediction model for regression test selection," *IEEE Transactions on Software Engineering*, vol. 27, no. 3, pp 248-263, Mar. 2001.

27. M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proc. of the 16th International Conference on Software Engineering*, pp.191-200, May 1994.

28. IPL Information Processing Ltd. "Cantata++ for Testing C, C++ and Java," http://www.ipl.com/products/tools/pt400.uk.php. Accessed: Nov. 13, 2006.

29. IBM Rational Software. *IBM Rational Test RealTime User's Guide, Windows and UNIX*, Version: 2003.06.13.

30. J. A. Jones and M. J. Harrold. "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 195-209, Mar. 2003.

31. J. M. Kim and A. Porter. "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proc. of the 24the International Conference on Software Engineering*, pp. 119-129, May 2002.

32. J. M. Kim, A. Porter, and G. Rothermel. "An empirical study of regression test application frequency," in *Proc. of the 22nd International Conference on Software Engineering*, pp. 126-135, Jun. 2000.

33. D. C. Kung, J. Gao, and P. Hsia, "Class firewall, test order, and regression testing of object-oriented Programs," *Journal of Object-Oriented Programming*, pp. 51-65, May 1995.

34. H. K. N. Leung and L. White. "A cost model to compare regression test strategies," in *Proc. of the Conference on Software Maintenance*, pp. 201-208, Oct. 1991.

35. H. K. N. Leung and L. White. "A study of integration testing and software regression at the integration level," in *Proc. of the Conference on Software Maintenance,* pp. 290-301, Nov. 1990.

36. H. K. N. Leung and L. White. "Insights into regression testing," in *Proc. of the Conference on Software Maintenance*, pp. 60-69, Oct. 1989.

37. A. G. Malishevsky, G. Rothermel, and S. Elbaum. "Modeling the cost-benefits tradeoffs for regression testing techniques," in *Proc. of the International Conference on Software Maintenance*, pp. 204-213, Oct. 2002.

38. B. Marick. "How to misuse code coverage," in *Proc. of the 16th International Conference on Testing Computer Software*, Jun. 1999.

39. A. P. Mathur. *Foundations of Software Testing*, Draft v3.0, Aug. 2006, Volume 1 under publication by Pearson Education.

40. J. C. Munson, A. P. Nikora, J. S. Sherif. "Software faults: a quantifiable definition," *Advances in Engineering Software*, vol. 37, no. 5, pp. 327-333, May 2006.

41. J. Offutt, J. Pan, and J. Voas. "Procedures for reducing the size of coverage-based test sets," in *Proc. of the 12th International Conference on Testing Computer Software*, pp. 111-123, Jun. 1995.

42. A. K. Onoma, W. T. Tsai, M. H. Poonawala, and H. Suganuma. "Regression testing in an industrial environment," *Communications of the ACM*, vol. 41, no. 5, pp. 81-86, May 1998.

43. A. Orso, N. Shi, and M. J. Harrold. "Scaling regression testing to large software systems," in *Proc. of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 241-251, Nov. 2004.

44. T. J. Ostrand, and M. J. Balcer. "The category-partition method for specifying and generating fuctional tests," *Communications of the ACM*, vol. 31, no. 6, pp. 676-686, Jun. 1988.

45. D. S. Rosenblum and E. J. Weyuker. "Predicting the cost-effectiveness of regression testing strategies," in *Proc. of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 118-135, Oct. 1996.

46. G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu. "On test suite composition and cost-effective regression testing," *ACM Transactions on Software Engineering and Methodology*, vol. 13, no. 3, pp. 277-331, Jul. 2004.

47. G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia. "The impact of test suite granularity on the cost-effectiveness of regression testing," in *Proc. of the 24th International Conference on Software Engineering*, pp. 130-140, May 2002.

48. G. Rothermel and M. J. Harrold. "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 2, pp. 173-210, Apr. 1997.

49. G. Rothermel and M. J. Harrold. "Analyzing regression test selection techniques," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 529-551, Aug. 1996.

50. G. Rothermel and M. J. Harrold. "Empirical studies of a safe regression test selection technique," *IEEE Transactions on Software Engineering*, vol. 24, no. 6, pp. 401-419, Jun. 1998.

51. G. Rothermel and M. J. Harrold. "Selecting regression tests for object-oriented software," in *Proc. of the International Conference on Software Maintenance*, pp. 14-25, Sep. 1994.

52. G. Rothermel, M. J. Harrold, and J. Dedhia. "Regression test selection for C++ software," *Journal of Software Testing, Verification, and Reliability*, vol. 10, no. 2, pages 77-109, Jun. 2000.

53. G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. "An empirical study of the effects of minimization on the fault detection capabilities of test suites," in *Proc. of the International Conference on Software Maintenance*, pp. 34-43, Nov. 1998.

54. G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong. "Empirical studies of test-suite reduction," *Journal of Software Testing, Verification, and Reliability*, vol. 12, no. 4, pp. 219-249, Dec. 2002.

55. G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929-948, Oct. 2001.

56. G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. "Test case prioritization: An empirical study," in *Proc. of the IEEE International Conference on Software Maintenance*, pp. 179-188, Sep. 1999.

57. M. J. Rummel, G. M. Kapfhammer, and A. Thall. "Towards the prioritization of regression test suites with data flow information," in *Proc. of the 2005 ACM Symposium on Applied Computing*, pp. 1499-1504, Mar. 2005.

58. M. Skoglund and P. Runeson. "A case study of the class firewall regression test selection technique on a large scale distributed software system," in *Proc. of the International Symposium on Empirical Software Engineering*, pp. 74-83, Nov. 2005.

59. Software-artifact Infrastructure Repository. http://esquared.unl.edu/sir. Accessed: Apr. 26, 2007.

60. A. Srivastava and J. Thiagarajan. "Effectively prioritizing tests in development environment," in *Proc. of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 97-106, Jul. 2002.

61. Telcordia Technologies. *Telcordia Software Visualization and Analysis Toolsuite – User's Manual*, First Edition, http://xsuds.argreenhouse.com/manual.html, Jul. 1998.

62. F. I. Vokolos and P. G. Frankl. "Empirical evaluation of the textual differencing regression testing technique," in *Proc. of the International Conference on Software Maintenance*, pp. 44-53, Nov. 1998.

63. F. I. Vokolos and P. G. Frankl. "Pythia: a regression test selection tool based on textual differencing," in *Proc. of the 3rd International Conference on Reliability, Quality and Safety of Software-Intensive Systems*, pp. 3-21, May 1997.

64. L. J. White and H. K. N. Leung. "A firewall concept for both control-flow and data-flow in regression integration testing," in *Proc. of the Conference on Software Maintenance*, pp. 262-271, Nov. 1992.

65. W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. "Effect of test set minimization on fault detection effectiveness," *Software: Practice and Experience*, vol. 28, no. 4, pp. 347-369, Apr. 1998.

66. W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. "A study of effective regression testing in practice," in *Proc. of the Eighth International Symposium On Software Reliability Engineering*, pp. 264-274, Nov. 1997.

67. W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini. "Test set minimization and fault detection effectiveness: A case study in a space application," *The Journal of Systems and Software*, vol. 48, no. 2, pp. 79-89, Oct. 1999.

Appendix A Detailed Results of Previous Empirical Studies

Section A.1 contains the results of studies involving procedural programming languages. Table A-1 summarizes a selection of empirical studies and their conclusions. Table A-2 describes the programs used in the studies. Table A-3 summarizes the techniques used to generate test cases and fault data, where applicable, for selected procedural studies.

Section A.2 contains the results of studies involving object-oriented programming languages. Table A-4 summarizes a selection of empirical studies and their conclusions. Table A-5 describes the programs used in the studies. Table A-6 summarizes the techniques used to generate test cases and fault data, where applicable, for selected object-oriented studies.

A.1 Studies Involving Procedural Programming Languages

Table A-1 – Empirical Studies using Experimental Techniques with Procedural Programming Languages.

Empirical Study	Techniques Used	Subject Programs				Number of Tests
		Program Name	Number and Type of Versions Used (including base version)	Size	Fault Data Used	
[37]	*Selection Techniques:* - Retest-all - Modified non-core entity[a] *Reduction Techniques:* - No reduction - GHS reduction[a] *Prioritization:* - Random - Optimal - Func-total - Func-addtl	bash	10 sequential versions.	From 48292 LOC having 1494 functions to 65632 LOC having 1950 functions.	58 Between 3 and 9 faults seeded per version.	1168[k]

[47]	*Selection Techniques:* - Retest-all - Modified entity (TestTube)[a] - Modified non-core entity[a] *Reduction Techniques:* - No reduction - GHS reduction[a] *Prioritization:* - Random - Optimal - Func-addtl	bash	6 sequential versions.	From 48292 LOC having 1499 functions to 59010 LOC having 1712 functions.	41 (Between 4 and 10 faults seeded per version)	1168[k]
		emp-server	6 sequential versions.	From 67719 LOC having 1159 functions to 68782 LOC having 1173 functions.	49 (Between 9 and 10 faults seeded per version)	1985[m]
[46]	*Retest-all* *Selection Techniques:* - Retest-all - Modified entity (TestTube)[a] - Modified non-core entity[a] - Modification-focused minimization[a] *Reduction Techniques:* - No reduction - GHS reduction[a] *Prioritization:* - Optimal - Func-addtl - Func-diff-addtl	bash	10 sequential versions.	From 48292 LOC having 1494 functions to 65474 LOC having 1950 functions.	159 faults seeded across all versions of both programs (attempted to seed 10 per version).	1168[k]
		emp-server	10 sequential versions.	From 63014 LOC having 1188 functions to 64396 LOC having 1189 functions.		1985[m]
[15]	*Selection Techniques:* - Retest-all - Modified non-core entity[a] - Modification-focused minimization[a] *Reduction Techniques:* - GHS reduction[a] *Prioritization:* - Random - Optimal - Func-total - Func-addtl	bash	10 sequential versions.	From 48292 LOC having 1494 functions to 65632 LOC having 1950 functions.	Between 3 and 9 faults seeded per version.	1168 (base version)[k]
		flex	5 sequential versions.	From 8254 LOC having 139 functions to 10416 LOC having 166 functions.	Between 1 and 7 faults seeded per version.	525 (base version)[l]
		grep	5 sequential versions.	From 8164 LOC having 130 functions to 13282 LOC having 182 functions.	Between 1 and 4 faults seeded per version.	613 (base version)[l]
		gzip	6 sequential versions.	From 4744 LOC having 86 functions to 6582 LOC having 108 functions.	Between 2 and 5 faults seeded per version.	217 (base version)[l]

[21]	*Selection Techniques:* - Retest-all - Random - DejaVu - Dataflow-coverage-based - Modification-focused minimization[b]	totinfo	1	346 LOC	23 seeded faults	1054[l]
		schedule	1	299 LOC	9 seeded faults	2650[l]
		schedule2	1	297 LOC	10 seeded faults	2710[l]
		tcas	1	138 LOC	41 seeded faults	1608[l]
		printtok1	1	402 LOC	7 seeded faults	4130[l]
		printtok2	1	483 LOC	10 seeded faults	4115[l]
		replace	1	516 LOC	32 seeded faults	5542[l]
		space	1	6218 LOC	33 real faults	13585[n]
		player	6 versions (1 base version with 5 real non-sequential modified versions)	49316 LOC having 766 functions (base version)	Real faults used, though no number is given.	1033[o]
[3]	*Selection Techniques:* - Retest-all - Modified entity (TestTube)[c] - DejaVu	totinfo	1	346 LOC	23 seeded faults	1054[l]
		schedule1	1	299 LOC	9 seeded faults	2650[l]
		schedule2	1	297 LOC	10 seeded faults	2710[l]
		tcas	1	138 LOC	41 seeded faults	1608[l]
		printtok1	1	402 LOC	7 seeded faults	4130[l]
		printtok2	1	483 LOC	10 seeded faults	4115[l]
		replace	1	516 LOC	32 seeded faults	5542[l]
		space	1	6218 LOC having 136 functions	38 real faults	13520[n]
		player	6 versions (1 base version with 5 real non-sequential modified versions)	49316 LOC having 766 functions (base version)	None specified.	1033[o]
[32]	*Selection Techniques:* - Retest-all - Modification-focused minimization[d] - DejaVu - Modified entity (TestTube) - Random (selects 25%, 50%, or 75% the original test suite)	totinfo	1	346 LOC	12 seeded faults	1054[l]
		schedule	1	299 LOC	7 seeded faults	2650[l]
		schedule2	1	297 LOC	8 seeded faults	2710[l]
		tcas	1	138 LOC	12 seeded faults	1608[l]
		printtokens	1	402 LOC	7 seeded faults	4130[l]
		printtokens2	1	483 LOC	9 seeded faults	4115[l]
		replace	1	516 LOC	12 seeded faults	5542[l]
		space	1	6218 LOC	10 real faults	13585[n]
[65]	*Reduction Techniques:* - Random - *x*Suds minimization[e]	Cal	1	163 LOC	20 seeded faults	162[q]
		Checkeq	1	90 LOC	20 seeded faults	166[q]
		Col	1	274 LOC	30 seeded faults	156[q]
		Comm	1	144 LOC	15 seeded faults	754[q]
		Crypt	1	121 LOC	15 seeded faults	156[q]
		Look	1	135 LOC	15 seeded faults	193[q]
		Sort	1	842 LOC	23 seeded faults	997[q]
		Spline	1	289 LOC	13 seeded faults	700[q]
		Tr	1	127 LOC	12 seeded faults	870[q]
		Uniq	1	125 LOC	18 seeded faults	431[q]

[66]	*Selection Techniques:* - *x*Suds modification-based selection[f] *Reduction Techniques:* - *x*Suds minimization[g] *Prioritization:* - *x*Suds prioritization[h]	space	1	"about 10000 lines" of code	10 real faults	1000[p]
[17]	*Prioritization:* - Random - Optimal - Stmt-total - Stmt-addtl - Stmt-FEP-total - Stmt-FEP- addtl - Func-total - Func-addtl - Func-FEP-total - Func-FEP-addtl - Func-FI-total - Func-FI-addtl - Func-FI-FEP-total - Func-FI-FEP-addtl	tot_info	1	346 LOC	23 seeded faults	1052[l]
		schedule	1	299 LOC	9 seeded faults	2650[l]
		schedule2	1	297 LOC	10 seeded faults	2710[l]
		tcas	1	138 LOC	41 seeded faults	1608[l]
		print_tokens	1	402 LOC	7 seeded faults	4130[l]
		print_tokens2	1	483 LOC	10 seeded faults	4115[l]
		replace	1	516 LOC	32 seeded faults	5542[l]
		space	1	6218 LOC	35 real faults	13585[n]
[18]	*Prioritization:* - Random - Optimal - Stmt-total - Stmt-addtl - Stmt-FEP-total - Stmt-FEP-addtl - Func-total - Func- addtl - Func-FEP-total - Func-FEP-addtl - Func-FI-total - Func-FI-addtl - Func-FI-FEP-total - Func-FI-FEP-addtl - Func-diff-total - Func-diff-addtl - Func-diff-FEP-total - Func-diff-FEP-addtl	tot_info	1	346 LOC	23 seeded faults	1052[l]
		schedule	1	299 LOC	9 seeded faults	2650[l]
		schedule2	1	297 LOC	10 seeded faults	2710[l]
		tcas	1	138 LOC	41 seeded faults	1608[l]
		print_tokens	1	483 LOC	7 seeded faults	4130[l]
		print_tokens2	1	402 LOC	10 seeded faults	4115[l]
		replace	1	516 LOC	32 seeded faults	5542[l]
		space	1	6218 LOC	35 real faults	13585[n]
		grep	5 sequential versions.	First version contained 7451 LOC having 133 functions	Between 1 and 4 faults seeded per version.	613[l]
		flex	5 sequential versions.	First version contained 9153 LOC having 140 functions	Between 1 and 8 faults seeded per version.	525[l]
		QTB	6 sequential versions.	Over 300 KLOC across 2875 functions.	17 real faults (detectable by regression tests): from 1 to 8 per version	Baseline version "included 135 test cases."[r]

[56]	*Prioritization:* - Untreated - Random - Optimal - Stmt-total - Stmt-addtl - Branch-total - Branch-addtl - Stmt-FEP-total - Stmt-FEP-addtl	tot_info	1	346 LOC	23 seeded faults	1052[l]
		schedule	1	299 LOC	9 seeded faults	2650[l]
		schedule2	1	297 LOC	10 seeded faults	2710[l]
		tcas	1	138 LOC	41 seeded faults	1608[l]
		print_tokens	1	402 LOC	7 seeded faults	4130[l]
		print_tokens2	1	483 LOC	10 seeded faults	4115[l]
[55]	*Prioritization:* - Untreated - Random - Optimal - Stmt-total - Stmt-addtl - Branch-total - Branch-addtl - Stmt-FEP-total - Stmt-FEP-addtl	replace	1	516 LOC	i) 32 seeded faults ii) 9622 mutant faults	5542[l]
		schedule	1	299 LOC	i) 9 seeded faults ii) 2153 mutant faults	2650[l]
		schedule2	1	297 LOC	i) 10 seeded faults ii) 2822 mutant faults	2710[l]
		tcas	1	138 LOC	i) 41 seeded faults ii) 2876 mutant faults	1608[l]
		print_tokens	1	402 LOC	i) 7 seeded faults ii) 4030 mutant faults	4130[l]
		print_tokens2	1	483 LOC	i) 10 seeded faults ii) 4346 mutant faults	4115[l]
		replace	1	516 LOC	i) 32 seeded faults ii) 9622 mutant faults	5542[l]
		space	1	6218 LOC	i) 35 real faults ii) 132163 mutants generated	13585[n]
[16]	*Prioritization:* - Random - Stmt-addtl - Func- addtl - Func-FI-addtl	space	1 (10 faulty versions created from this)	6218 LOC	Each fault version averaged 3.3 real faults.	50 test suites created having 148 to 166 test cases[n]
[19]	*Prioritization:* - Random - Optimal - Func-total - Func-addtl - Func-diff-total - Func-diff-addtl	bash	10	49 KLOC (base version)	7.8 seeded faults per version (average)	2 test suites: One of size 1168[k], the other of size 19.
		emp-server	10	68 KLOC (base version)	10.0 seeded faults per version (average)	2 test suites: One of size 1985[l], the other of size 32.
		flex	5	8 KLOC (base version)	4.5 seeded faults per version (average)	567[l]
		grep	5	11 KLOC (base version)	2.8 seeded faults per version (average)	809[l]
		gzip	6	5 KLOC (base	11.8 seeded	217[l]

				version)	faults per version (average)	
		make	5	18 KLOC (base version)	4.0 seeded faults per version (average)	1043[l]
		sed	2	8 KLOC (base version)	4.5 seeded faults per version (average)	1293[l]
		xearth	3	24 KLOC (base version)	3.5 seeded faults per version (average)	539[l]
[48]	*Selection Technique:* - DejaVu	totinfo	1	440 LOC	23 seeded faults	1054[l]
		schedule1	1	292 LOC	9 seeded faults	2650[l]
		schedule2	1	301 LOC	10 seeded faults	2680[l]
		tcas	1	141 LOC	41 seeded faults	1578[l]
		usl.123 (aka printtok1)	1	472 LOC	7 seeded faults	4056[l]
		usl.128 (aka printtok2)	1	399 LOC	10 seeded faults	4071[l]
		replace	1	512 LOC	32 seeded faults	5542[l]
		player (dejavu simulated)	6 versions (1 base version with 5 real non-sequential modified versions)	49316 LOC having 766 functions (base version)	None specified.	1035[o]
[50]	*Selection Technique:* - DejaVu	totinfo	1	346 LOC	23 seeded faults	1052[l]
		schedule1	1	299 LOC	9 seeded faults	2650[l]
		schedule2	1	297 LOC	10 seeded faults	2710[l]
		tcas	1	138 LOC	41 seeded faults	1608[l]
		printtok1	1	402 LOC	7 seeded faults	4130[l]
		printtok2	1	483 LOC	10 seeded faults	4115[l]
		replace	1	516 LOC	32 seeded faults	5542[l]
		player (dejavu simulated)	6 versions (1 base version with 5 real non-sequential modified versions)	49316 LOC having 766 functions (base version)	None specified.	1033[o]
		Windows NT commercial program (dejavu simulated)	10 sequential versions.	2145 LOC having 27 functions (base version)	None specified.	388[r] (3 scripts implemented the tests)
[24]	*Reduction Technique:* - GHS reduction[i]	procedures: trityp, atof, getop, calc, qsort, sqroot	1 to 5 experimental versions	From 17 to 33 LOC	None specified.	From 2 to 19 tests.[s]
[53]	*Reduction Technique:* - GHS reduction[b]	totinfo	1	346 LOC	23 seeded faults	1052[l]
		schedule1	1	299 LOC	9 seeded faults	2650[l]
		schedule2	1	297 LOC	10 seeded faults	2710[l]
		tcas	1	138 LOC	41 seeded faults	1608[l]
		printtok1	1	402 LOC	7 seeded faults	4130[l]
		printtok2	1	483 LOC	10 seeded faults	4115[l]
		replace	1	516 LOC	32 seeded faults	5542[l]
[54]	*Reduction Techniques:* - Random minimization - GHS reduction[b]	totinfo	1	346 LOC	23 seeded faults	1052[l]
		schedule2	1	297 LOC	10 seeded faults	2710[l]
		tcas	1	138 LOC	41 seeded faults	1608[l]
		printtok1	1	402 LOC	7 seeded faults	4130[l]
		printtok2	1	483 LOC	10 seeded faults	4115[l]
		replace	1	516 LOC	32 seeded faults	5542[l]

		space	1	6218 LOC	35 real faults	13585[n]
[8]	*Selection Technique:* - Modified entity (TestTube)[j]	SFIO programming library	1 (67 different maintenance versions were simulated)	Roughly 11000 LOC	None specified.	39 test units Each test unit is a C program (with a main function) that makes a sequence of calls to the library and covers a subset of entities in library.
		incl	2 sequential versions.	Roughly 1700 LOC	None specified.	8 test units Each test unit is a UNIX shell script that invokes incl with a number of command line options.
[41]	*Reduction Technique:* - Offutt et al's coverage based test reduction	10 program units	1 version for each	Ranging in size from 10 to 48 executable statements.	Between 183 and 3010 mutants generated for each program unit.	Less than 50 per program unit. Generated using an automated test case generator.
[67]	*Reduction Techniques:* - Random - *x*Suds minimization[e]	space	1	About 6100 LOC	18 real faults	1000[p]

[a] using a function coverage criterion
[b] using a control-flow graph edge coverage criterion
[c] using the following coverage criteria: function definitions, global variables, types, and preprocessor definitions
[d] using a control-flow graph node coverage criterion
[e] using block, decision, and all-uses coverage criteria respectively
[f] the coverage criteria used are not specified
[g] using a block coverage criterion
[h] using block coverage with feedback
[i] using an "all-uses" criterion
[j] using the following coverage criteria: functions, variables, preprocessor macros, types and files
[k] Comprised of test cases accompanying the software release and additional test cases created by the experimenters to exercise uncovered functionality.
[l] The test suite was designed by experimenters initially using the category partition method. Additional test cases were added to increase code coverage.
[m] The test suite was designed by experimenters using the category partition method.
[n] Test cases initially generated randomly by Vokolos and Frankl [62]. Experimenters then added new test cases until each executable edge in the program's control flow graph was exercised by at least 30 test cases.
[o] Experimenters created test cases using Empire information files as an informal specification and believed the resulting test suite to be "typical of the sorts of functional test suites designed in practice for large software systems".
[p] A pool of test cases was "created based on the operational profile of how the space program was used".
[q] A simple random domain-based test generator was implemented.
[r] Test cases designed by the software producer.
[s] Test cases designed by graduate students.

Table A-2 – Descriptions of Programs used in Previous Empirical Studies.

Program	Description
tcas	Altitude separation.[a]
shedule2	Priority scheduler.[a]
schedule	Priority scheduler.[a]
replace	Pattern replacement.[a]
tot_info	Information measure.[a]
print_tokens2	Lexical analyzer.[a]
print_tokens	Lexical analyzer.[a]
space	Language interpreter. Used by the European Space Agency.
gzip	Data compression / decompression.
sed	Stream editor.
xearth	View earth from space.
flex	Lexical analyzer generator.
grep	Pattern search utility.
make	Build manager.
QTB	An embedded real-time subsystem that performs initialization tasks on a level-5 RAID storage system.
bash	UNIX shell. An open-source application that provides a command line interface to multiple UNIX services.
emp-server	Server component of the open-source client-server internet game Empire. [46]
player	Subsystem from Internet game Empire. The five versions are actual modifications made during the maintenance of the software. Each version contains between 50 and 700 lines of modified code.
commercial	Windows NT application
SFIO library	Programming library SFIO (Safe, Fast Input/Output) – replacement for standard UNIX I/O library STDIO.
incl	Source Code Analyzer. A program for detecting and removing unnecessary #include directives from C programs.
Cal	Print a calendar. Unix utility.
Checkeq	Report missing or unbalanced delimiters. Unix utility.
Col	Filter reverse paper motions. Unix utility.
Comm	Select or reject lines common to two sorted files. Unix utility.
Crypt	Encrypt or decrypt a file. Unix utility.
Look	Find words in system directory or lines in a sorted list. Unix utility.
Sort	Sort and merge files. Unix utility.
Spline	Interpolate smooth curve based on given data. Unix utility.
Tr	Translate characters. Unix utility.
Uniq	Report or remove adjacent duplicate lines. Unix utility.
trityp	Procedure that classifies triangles as to their types.
atof	C procedure that converts character data to floating point data.
getop	Routine for a calculator program.
calc	Routine for a calculator program.
qsort	Quicksort routine.
sqroot	Program to calculate square root.

[a] One of the "Siemens programs". Chosen to be "large and complex enough to be considered realistic, and to permit the seeding of many hard-to-find errors... The programs used for the experiment are C programs obtained from various sources." [27]

Table A-3 – Techniques used to Generate Test Cases and Fault Data for Programs used in Selected Procedural Studies.

Program	How were test cases created?	Source of fault data.
tcas	For each base program, Black-box test cases were initially written, using category partitioning method and the Siemens Test Specification language tool. This test suite was then augmented with manually created white-box test cases. They ensured that each statement, edge, and definition-use pair in base program or its control flow graph was exercised by at least 30 test cases. [55]	Siemens researchers created faulty versions of these base programs by manually seeding them with faults, usually by modifying a single line of code. Ten people performed the fault seeding. [55]
schedule2		
schedule		
replace		
tot_info		
print_tokens2		
print_tokens		

space	An initial test pool of 10000 was generated randomly by Vokolos and Frankl. For this study additional tests were added after the program was instrumented for coverage. The researchers instrumented the program for coverage and ensured that each statement and edge was exercised by at least 30 test cases. [55]	Rothermel et al. created 35 faulty versions derived from the base version. 33 faults were found in development (3 of these were later dismissed as they were semantically equivalent to base version). 5 additional faults were found through working with the program. [55]
player [48]	Used Empire information files to construct realistic test suite. Treated the information files as informal specifications. Wrote functional tests for each command. Test suite for player contains black box test cases. Constructed from Empire information files (informal specifications).	--
bash [19]	A test suite was provided with Bash 2.0. To exercise functionality not covered by original test suite, additional test cases were created by considering reference documentation for bash as informal specification. Resulting test cases exercise an average 64% of functions across all versions of system. [37] bash had been released with test suites composed of test cases from previous versions and new test cases designed to validate added functionality. Additional work was done to update the available test suite for testing across the different versions.	bash: 3 to 9 faults seeded per version. Added by graduate and undergraduate students with 2 years experience coding in C [37]. bash and emp-server seeded with 10 faults each per version by undergrad and grad students. Regression faults that could not be detected by any test cases were discarded. Regression faults that were detected by more than 80% of test cases were discarded – the assumption being that such easily detected faults would be detected by test engineers during testing of modifications. [46, 47] In [18], 4 versions of grep and flex were seeded with at least 20 faults by graduate students. (A baseline version for each of grep and flex is also available.) Only faults that were exposed by at least 1 and at most 20 per cent of tests were kept for the study. (Between 1 and 8 faults remained for each of the 8 faulty program versions.) For all: No real faults available. Therefore undergraduate and graduate students with at least two years of C programming experience were instructed to insert realistic faults into the code. The faults had to involve code deleted from, inserted into, or modified between the versions. Ten faults were seeded into each version of the program. Faults that were not detected or that were detected by 25% or more of the test cases were excluded. The assumption being that such faults would be easily detected by engineers during unit testing. The average number of faults remaining in programs across all versions is demonstrated in the above table. [19] In [15] bash, flex, grep, gzip were used. Two undergraduate students created 10 faults per version. Faults not detected and those detected by more than 25% of test cases were discarded.
emp-server flex grep gzip make sed xearth	No test cases available. Used program documentation (informal specifications) together with category partition method and code coverage tools at statement level to increase code coverage. Created test suites for base versions of programs to serve as regression suites for subsequent versions. [19]	
Cal, Checkeq, Colm, Comm, Crypt, Look, Sort, Spline, Tr, and Uniq [65]	A simple random domain-based test generator was implemented for each of the programs. According to the command line syntax of the program under test, random strings meeting the input signature are generated. Full coverage was not generally achieved, as the test cases were not manually tuned to achieve 100% coverage.	Faults injected by graduate students into each program. One erroneous program was created per fault. Twelve to thirty faulty versions are created for the programs.
trityp, atof, getop, calc, qsort, and sqroot [24]	Graduate students developed the test cases. Functional test cases were first written to test the program and then test cases were developed to satisfy any uncovered definition-use pairs.	--

A.2 Studies Involving Object-Oriented Programming Languages

Table A-4 – Empirical Studies using Experimental Techniques with Object-Oriented Programming Languages.

Empirical Study	Techniques used	Subject programs				Number of Tests
		Program Name	Number and Type of Versions Used (including base version)	Size	Fault Data Used	
[12]	*Prioritization:* - Untreated - Random - Optimal - Block-total - Block-addtl - Method-total - Method-addtl - Method-diff-total - Method-diff-addtl	Ant	9 sequential versions.	627 classes (most recent version)	21 seeded faults across all versions	150 test classes. 877 test methods[a]
		XML-security	4 sequential versions.	143 classes (most recent version)	6 seeded faults across all versions	14 test classes, 83 test methods[a]
		JMeter	6 sequential versions.	389 classes (most recent version)	9 seeded faults across all versions	28 test classes, 78 test methods[a]
		JTopas	4 sequential versions.	50 classes (most recent version)	5 seeded faults across all versions	11 test classes, 128 test methods[a]
[11]	*Prioritization:* - Untreated - Random - Optimal - Block-total - Block-addtl - Method-total - Method-addtl	Ant	9 sequential versions.	627 classes (most recent version)	2907 mutants generated across all versions	150 test classes. 877 test methods[a]
		XML-security	4 sequential versions.	143 classes (most recent version)	127 mutants generated across all versions	14 test classes, 83 test methods[a]
		JMeter	6 sequential versions.	389 classes (most recent version)	295 mutants generated across all versions	28 test classes, 78 test methods[a]
		JTopas	4 sequential versions.	50 classes (most recent version)	8 mutants generated across all versions	11 test classes, 128 test methods[a]
[25]	*Selection Techniques:* - DejaVOO	SIENA	7 sequential versions.	185 methods (original version)	None specified.	138[a]
		JEdit	2 real sequential versions. (11 sequential versions are created based upon the changes made)	3495 methods (original version)	None specified.	189[b]
		JMeter	2 real sequential versions. (5 sequential versions are created based upon the changes made).	109 methods (original version)	None specified.	50[c]
		RegExp	2 real sequential versions. (10 sequential	168 methods (original version)	None specified.	66[a]

			versions are created based upon the changes made)			
[43]	*Selection Techniques:* - DejaVOO	Jaba	5 sequential versions.	70 KLOC, 525 classes and interfaces (average across versions)	None specified.	707[a]
		Daikon	5 sequential versions.	167 KLOC, 824 classes and interfaces (average across versions)	None specified.	200[a]
		JBoss	5 sequential versions.	532 KLOC, 2403 classes and interfaces (average across versions)	None specified.	639[a]
[58]	*Selection Techniques:* - class firewall technique - change based technique (test case is selected if it executes a changed class)	large distributed component J2EE system – banking software (3 applications in the system are evaluated)	3 sequential versions.	The entire system consists of more than 1.2 MLOC having approximately 27000 classes	None specified.	51[d]
[60]	*Prioritization:* - Echelon/Scout	3 Unidentified Programs	2 real sequential versions of each program.	For the three programs: 31020, 1761, and 1967 functions are used in the first of the two versions	None specified.	For the three programs: 3128, 56, and 56 test cases are used respectively These are tests used in actual development.
[52]	*Selection Techniques:* - Control-flow-based technique	Tools.h++	6 sequential versions.	24849 LOC with 186 classes (most recent)	None specified.	61 test cases (coarse granularity), or between 281 to 317 test cases (fine granularity) depending on the version[f]
[57]	*Prioritization:* - An all-DU test adequacy criterion is used to prioritize the test suite.	Bank[e]	1	1 class, 5 methods	4	7 test cases[a]
		Identifier[e]	1	3 classes, 13 methods	6	11 test cases[a]
		Money[e]	1	3 classes, 33 methods	9	21 test cases[a]

[a] Test cases were obtained from the object's distribution.
[b] Experimenters created a test suite "to exercise various features of the text editor."
[c] "Created by considering combinations of features available through the user interface of the system."
[d] "Regression tests are performed by the test engineers who follow scenario procedures for each application where various tasks are performed in the application."
[e] These programs were written for educational purposes.
[f] Test cases created by the developer.

Table A-5 – Descriptions of Programs used in Previous Object-Oriented Studies.

Program	Description
Tools.h++ library [52]	Commercial C++ class library.
SIENA [25]	Internet-based event notification system.
JEdit [25]	Text editor.
JMeter [25]	Web-applications testing tool.
RegExp [25]	Regular-expression library.
Jaba [43]	Framework for analyzing Java programs.
Daikon [43]	A tool that performs dynamic invariant detection.
JBoss [43]	Fully-feature Java application server.
Ant [12, 11]	Java-based build tool similar to make.
XML-security [12, 11]	Implements security standards for XML.
JMeter [12, 11]	Web-applications testing tool.
JTopas [12, 11]	Used for parsing text data.
Bank [57]	Written by computer science professor for educational purposes. It simulates possible actions that can be performed on a bank account system.
Identifier [57]	Written by undergraduate students.
Money [57]	Provided with JUnit framework.

Table A-6 – Techniques used to Generate Test Cases and Fault Data for Programs used in Selected Object-Oriented Studies.

Program	How were test cases created?	Source of fault data.
SIENA [25]	Test suite created by program developers. The study's authors added test cases to increase coverage at the method level. 70% method coverage achieved.	--
JEdit [25]	Obtained two successive development versions – 11 versions were created based on changes between releases. The study's authors developed a test suite. 75% method coverage achieved.	--
JMeter [25]	Obtained two successive development versions – 5 versions were created based on changes between releases. Test suite had to be developed by the study's authors. The test cases were developed by considering combinations of features available through the user interface of the system. 67% method coverage achieved.	--
RegExp [25]	Obtained two successive development versions – 10 versions were created based on changes between releases. Test cases provided with library. 46% method coverage achieved.	--
Jaba [43]	Test suites created by programs' developers. For each case, 5 consecutive versions were obtained.	--
Daikon [43]		
JBoss [43]		

Ant [12, 11]	Test cases are available from each object's distribution. The studies' authors created test scripts to execute and validate test cases automatically.	Object programs were not provided with any faults or fault data. Two graduate students seeded faults. Involved code inserted into or modified in each of the versions. Faults that were detected by more than 20% of test cases were excluded.
XML-security [12, 11]		
JMeter [12, 11]		
JTopas [12, 11]		
Bank [57]	JUnit test suite provided with each application.	Faults were manually seeded into program by graduate students.
Identifier [57]		
Money [57]		

Appendix B *x*Suds Commands and Experimental Methodology

B.1 The *x*Suds Command Set

The Telcordia Software Visualization and Analysis Toolsuite manual [61] summarizes the full range of the commands and options available using the *x*Suds software. A subset of the commands summarizing only the functionality necessary in the context of the experiments performed for this study is described below.

B.1.1 atacCL (for Windows only)

The atacCL command compiles and links C and C++ programs. Data-flow files (ending in .atac) are created for each C and C++ source file compiled. For example, compiling file source.c results in the creation of the corresponding data-flow file source.atac.

A resulting executable program is instrumented such that when it is run coverage traces are outputted to a trace file (ending in the .trace extension). The default trace filename is named atac.trace but the user may specify an alternate name. A trace file contains the coverage information for each of the test cases run against a program version under study.

B.1.2 atacdiff

The atacdiff command creates a file (ending in .dif) that encodes the differences between two given versions of a source file. For example, given two files, source.c and sourceNew.c, atacdiff generates a file source.dif containing the difference information.

B.1.3 atac

SYNOPSIS

atac [OPTIONS] [trace-file (*.trace)] [data-flow files (*.atac)] [difference files (*.dif) (optional)]

DESCRIPTION

The atac command determines code coverage by analyzing the data-flow files with the trace file produced by the test executions of the program under study. By using code difference information between the program's source files and the modified source files of a successive version, it is possible to restrict the analysis to only source code that has been modified.

OPTIONS

-t specifies that a coverage summary is to be presented on a per test case basis – non-zero only

-p specifies that a coverage summary is to be presented on a per test case basis

-m {erBbd}

Specifies the coverage measure(s) to be used: e, r, B, b, and d can be used to specify function entry, function return, block, basic block and decision coverage respectively.

-M presents a minimal set of test cases that achieves the same coverage, for the specified coverage measure, as all cases together.

-Q sorts the test cases in order of additional coverage.

B.1.4 atactm

The atactm command is used to manage the trace files created when executing a program created by atacCL. In the experimental analysis, the atactm command is used to extract the coverage data associated with selected test cases in order to modify existing trace files and to create new trace files.

B.2 Conducting the Experimental Analysis

Sections B.2.1 through B.2.7 describe how each of the experimental analyses is

conducted using *x*Suds. Section B.2.8 describes how a trace file can be randomly ordered.

B.2.1 Modification-based regression test selection

*x*Suds command required:

```
atac -t -m{erBbd} [trace-file (*.trace)] [data-flow files (*.atac)] [difference files (*.dif)]
```

Only the data-flow files from the original version that have been modified or deleted are inputted. The difference files containing the differences between the original version and the modified version are inputted. The trace file contains the coverage information for each of the test cases run against the original version.

B.2.2 Modification-focused minimization

*x*Suds command required:

```
atac -M -m{erBbd} [trace-file (*.trace)] [data-flow files (*.atac)] [difference files (*.dif)]
```

Only the data-flow files from the original version that have been modified or deleted are inputted. The difference files containing the differences between the original version and the modified version are inputted. The trace file contains the coverage information for each of the test cases run against the original version.

Section B.2.8 explains how the trace file data should be randomly ordered to properly assess the effectiveness of this technique.

B.2.3 Coverage-focused minimization

*x*Suds command required:

```
atac -M -m{erBbd} [trace-file (*.trace)] [data-flow files (*.atac)]
```

All the data-flow files from the original version are inputted. The trace file contains the coverage information for each of the test cases run against the original version.

Section B.2.8 explains how the trace file data should be randomly ordered to properly

assess the effectiveness of this technique.

B.2.4 Total-coverage prioritization

STEP 1

*x*Suds command required:

```
atac -p -m{erBbd} [trace-file (*.trace)] [data-flow files (*.atac)]
```

All the data-flow files from the original version are inputted. The trace file contains the coverage information for each of the test cases run against the original version.

STEP 2

Given the output generated from the step above, sort the list according to the coverage achieved on a per test case basis. If multiple test cases provide the same coverage, they are ordered randomly.

B.2.5 Additional-coverage prioritization

The *x*Suds prioritization command does not match the described algorithm as found in Table 2-8 (see the M9 description). In the case of M9, once all function entries have been covered, the prioritization process does not repeat on the remaining test cases. Instead, the prioritization stops. The rest of the test cases are essentially listed in reverse order of their original sequence.

Correct for this by repeating the prioritization command on any remaining unprioritized test cases until all the test cases have been prioritized as prescribed by the technique.

STEP 1

*x*Suds command required:

```
atac -Q -m{erBbd} [trace-file (*.trace)] [data-flow files (*.atac)]
```

All the data-flow files from the original version are inputted. The trace file initially

contains the coverage information for each of the test cases run against the original version; its contents might be modified according to the instructions of STEP 2.

Section B.2.8 explains how the trace file data should be randomly ordered to properly assess the effectiveness of this technique.

STEP 2

If the list of test cases is only partially prioritized in STEP 1 then adjust the trace file using the atactm command to remove those test cases that have been prioritized and return to STEP 1. If all test cases have been prioritized, then the prioritization is complete.

B.2.6 Total-coverage-using-modification-information prioritization

STEP 1

*x*Suds command required:

atac -p -m{erBbd} [trace-file (*.trace)] [data-flow files (*.atac)] [difference files (*.dif)]

Only the data-flow files from the original version that have been modified or deleted are inputted. The difference files containing the differences between the original version and the modified version are inputted. The trace file contains the coverage information for each of the test cases run against the original version.

STEP 2

Given the output generated from the step above, sort the list according to the coverage achieved per test case. If multiple test cases provide the same coverage, they are ordered randomly.

STEP 3

Apply Total-coverage prioritization to any test cases that remain unsorted by this technique.

B.2.7 Additional-coverage-using-modification-information prioritization

The *x*Suds prioritization command does not match the described algorithm as found in Table 2-8 (see the M19 description). In the case of M19, once all function entries have been covered, the prioritization process does not repeat on the remaining test cases. Instead, the prioritization stops. The rest of the test cases are essentially listed in reverse order of their original sequence.

Correct for this by repeating the prioritization command on any remaining unprioritized test cases until all the test cases have been prioritized as prescribed by the technique.

STEP 1

*x*Suds command required:

atac -Q -m{erBbd} [trace-file (*.trace)] [data-flow files (*.atac)] [difference files (*.dif)]

Only the data-flow files from the original version that have been modified or deleted are inputted. The difference files containing the differences between the original version and the modified version are inputted. The trace file initially contains the coverage information for each of the test cases run against the original version; its contents might be modified according to the instructions of STEP 2.

Section B.2.8 explains how the trace file data should be randomly ordered to properly assess the effectiveness of this technique.

STEP 2

If the list of test cases is only partially prioritized in STEP 1 then adjust the trace file using the atactm command to remove those test cases that have been prioritized and return to STEP 1. If all test cases covering the modified code have been prioritized, then proceed to STEP 3.

STEP 3

Apply Additional-coverage prioritization to any test cases that remain unsorted by this

technique.

B.2.8 Randomizing the trace-file

As discussed in section 3.7.5, the *x*Suds software does not perform random selection as prescribed for the prioritization techniques. For example, according to technique M9, "if multiple test cases cover the same number of function entries not yet covered, they are ordered randomly". This is not done by the atac command; instead the tool always selects the first test case it finds in the trace file listing. (When the test cases are run against a program compiled by atacCL, the coverage information is added sequentially to the resulting trace file in the order the test cases are run.)

In order to achieve the randomness desired in the prioritization selections, the initial trace file resulting from the test case executions can be reordered randomly using the atactm command to create a new trace file. By using this newly created as trace file (containing the same coverage information as the initial trace file) with the atac command, the desired random selection will now occur.

The same concern exists for the minimization technique as for the prioritization technique. The atac command makes minimization selections according to the test case execution order as listed in the trace files.

Appendix C Detailed Experimental Results

C.1 Modification-Focused Minimization

The following tables contain the detailed results of the modification-focused minimization executions for each coverage criterion used. As explained in section 3.7.5, each of the minimizations is executed against 100 randomly ordered trace files for each of the software versions. For each coverage criterion, for each software version, and for all the software versions combined (to get average values across all versions – denoted by 'All' in the tables), the *percentage reduction in test suite size* and the *percentage reduction in fault detection effectiveness* metrics are calculated. The median, minimum (min), maximum (max), mean, standard deviation (stdev), 2.5th percentile, and 97.5th percentile values are recorded.

Table C-1 – Detailed Results for the All-Techniques-Combined Coverage.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5th percentile)	Upper Bound (97.5th percentile)
3	70.9 (101)	70.9 (101)	70.9 (101)	70.9 (101)	0 (0)	70.9 (101)	70.9 (101)
4	87.8 (73)	87.8 (73)	87.8 (73)	87.8 (73)	0 (0)	87.8 (73)	87.8 (73)
5	80.6 (146)	80.6 (146)	80.6 (146)	80.6 (146)	0 (0)	80.6 (146)	80.6 (146)
6	83.0 (144)	83.0 (144)	83.0 (144)	83.0 (144)	0 (0)	83.0 (144)	83.0 (144)
All	81.8 (464)	81.8 (464)	81.8 (464)	81.8 (464)	0 (0)	81.8 (464)	81.8 (464)
Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5th percentile)	Upper Bound (97.5th percentile)
3	20.0 (24)	16.7 (25)	23.3 (23)	19.4 (24.19)	1.9 (0.56)	16.7 (25)	23.3 (23)
4	80.0 (1)	60.0 (2)	100 (0)	82.0 (0.9)	12.9 (0.64)	60.0 (2)	100 (0)
5	0 (1)	0 (1)	0 (1)	0 (1)	0 (0)	0 (1)	0 (1)
6	0 (1)	0 (1)	0 (1)	0 (1)	0 (0)	0 (1)	0 (1)
All	27.0 (27)	21.6 (29)	32.4 (25)	26.8 (27.09)	2.5 (0.92)	21.6 (29)	31.1 (25.48)

Table C-2 – Detailed Results for the Function-Entry Coverage.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	96.5 (12)	96.5 (12)	96.5 (12)	96.5 (12)	0 (0)	96.5 (12)	96.5 (12)
4	97.3 (16)	97.3 (16)	97.3 (16)	97.3 (16)	0 (0)	97.3 (16)	97.3 (16)
5	97.7 (17)	97.7 (17)	97.7 (17)	97.7 (17)	0 (0)	97.7 (17)	97.7 (17)
6	97.8 (19)	97.8 (19)	97.8 (19)	97.8 (19)	0 (0)	97.8 (19)	97.8 (19)
All	97.5 (64)	97.5 (64)	97.5 (64)	97.5 (64)	0 (0)	97.5 (64)	97.5 (64)

Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	100 (0)	100 (0)	100 (0)	100 (0)	0 (0)	100 (0)	100 (0)
4	100 (0)	100 (0)	100 (0)	100 (0)	0 (0)	100 (0)	100 (0)
5	100 (0)	100 (0)	100 (0)	100 (0)	0 (0)	100 (0)	100 (0)
6	100 (0)	100 (0)	100 (0)	100 (0)	0 (0)	100 (0)	100 (0)
All	100 (0)	100 (0)	100 (0)	100 (0)	0 (0)	100 (0)	100 (0)

Table C-3 – Detailed Results for the Function-Return Coverage.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	96.8 (11)	96.8 (11)	96.8 (11)	96.8 (11)	0 (0)	96.8 (11)	96.8 (11)
4	96.2 (23)	96.2 (23)	96.2 (23)	96.2 (23)	0 (0)	96.2 (23)	96.2 (23)
5	97.5 (19)	97.5 (19)	97.5 (19)	97.5 (19)	0 (0)	97.5 (19)	97.5 (19)
6	97.1 (25)	97.1 (25)	97.1 (25)	97.1 (25)	0 (0)	97.1 (25)	97.1 (25)
All	96.9 (78)	96.9 (78)	96.9 (78)	96.9 (78)	0 (0)	96.9 (78)	96.9 (78)

Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	100 (0)	96.7 (1)	100 (0)	98.4 (0.47)	1.7 (0.50)	96.7 (1)	100 (0)
4	100 (0)	80.0 (1)	100 (0)	91.2 (0.44)	10.0 (0.50)	80.0 (1)	100 (0)
5	100 (0)	100 (0)	100 (0)	100 (0)	0 (0)	100 (0)	100 (0)
6	100 (0)	100 (0)	100 (0)	100 (0)	0 (0)	100 (0)	100 (0)
All	97.3 (1)	94.6 (2)	100 (0)	97.5 (0.91)	2.1 (0.77)	94.6 (2)	100 (0)

Table C-4 – Detailed Results for the Block Coverage.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	72.9 (94)	72.9 (94)	72.9 (94)	72.9 (94)	0 (0)	72.9 (94)	72.9 (94)
4	89.1 (65)	89.1 (65)	89.1 (65)	89.1 (65)	0 (0)	89.1 (65)	89.1 (65)
5	83.0 (128)	83.0 (128)	83.0 (128)	83.0 (128)	0 (0)	83.0 (128)	83.0 (128)
6	85.6 (122)	85.6 (122)	85.6 (122)	85.6 (122)	0 (0)	85.6 (122)	85.6 (122)
All	84.0 (409)	84.0 (409)	84.0 (409)	84.0 (409)	0 (0)	84.0 (409)	84.0 (409)

Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	23.3 (23)	20.0 (24)	26.7 (22)	22.7 (23.19)	1.9 (0.56)	20.0 (24)	26.7 (22)
4	80.0 (1)	80.0 (1)	100 (0)	85.8 (0.71)	9.1 (0.46)	80.0 (1)	100 (0)
5	0 (1)	0 (1)	0 (1)	0 (1)	0 (0)	0 (1)	0 (1)
6	0 (1)	0 (1)	0 (1)	0 (1)	0 (0)	0 (1)	0 (1)
All	29.7 (26)	27.0 (27)	35.1 (24)	30.0 (25.9)	2.2 (0.81)	27.0 (27)	35.1 (24)

Table C-5 – Detailed Results for the Basic-Block Coverage.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	72.9 (94)	72.9 (94)	72.9 (94)	72.9 (94)	0 (0)	72.9 (94)	72.9 (94)
4	89.1 (65)	89.1 (65)	89.1 (65)	89.1 (65)	0 (0)	89.1 (65)	89.1 (65)
5	83.0 (128)	83.0 (128)	83.0 (128)	83.0 (128)	0 (0)	83.0 (128)	83.0 (128)
6	85.6 (122)	85.6 (122)	85.6 (122)	85.6 (122)	0 (0)	85.6 (122)	85.6 (122)
All	84.0 (409)	84.0 (409)	84.0 (409)	84.0 (409)	0 (0)	84.0 (409)	84.0 (409)

Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	23.3 (23)	20.0 (24)	26.7 (22)	22.7 (23.19)	1.9 (0.56)	20.0 (24)	26.7 (22)
4	80.0 (1)	80.0 (1)	100 (0)	85.8 (0.71)	9.1 (0.46)	80.0 (1)	100 (0)
5	0 (1)	0 (1)	0 (1)	0 (1)	0 (0)	0 (1)	0 (1)
6	0 (1)	0 (1)	0 (1)	0 (1)	0 (0)	0 (1)	0 (1)
All	29.7 (26)	27.0 (27)	35.1 (24)	30.0 (25.9)	2.2 (0.81)	27.0 (27)	35.1 (24)

Table C-6 – Detailed Results for the Decision Coverage.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	76.1 (83)	76.1 (83)	76.1 (83)	76.1 (83)	0 (0)	76.1 (83)	76.1 (83)
4	88.8 (67)	88.8 (67)	88.8 (67)	88.8 (67)	0 (0)	88.8 (67)	88.8 (67)
5	83.3 (126)	83.3 (126)	83.3 (126)	83.3 (126)	0 (0)	83.3 (126)	83.3 (126)
6	83.9 (137)	83.9 (137)	83.9 (137)	83.9 (137)	0 (0)	83.9 (137)	83.9 (137)
All	83.8 (413)	83.8 (413)	83.8 (413)	83.8 (413)	0 (0)	83.8 (413)	83.8 (413)
Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	20.0 (24)	16.7 (25)	23.3 (23)	20.4 (23.88)	2.3 (0.69)	16.7 (25)	23.3 (23)
4	80.0 (1)	60.0 (2)	100 (0)	82.0 (0.9)	12.9 (0.64)	60.0 (2)	100 (0)
5	0 (1)	0 (1)	0 (1)	0 (1)	0 (0)	0 (1)	0 (1)
6	0 (1)	0 (1)	0 (1)	0 (1)	0 (0)	0 (1)	0 (1)
All	27.0 (27)	21.6 (29)	32.4 (25)	27.6 (26.78)	2.7 (1.0)	21.6 (29)	32.4 (25)

C.2 Coverage-Focused Minimization

The following tables contain the detailed results of the coverage-focused minimization executions for each coverage criterion used. As explained in section 3.7.5, each of the test suite reductions is executed against 100 randomly ordered trace files for each of the software versions. For each coverage criterion, for each software version, and for all the software versions combined (to get average values across all versions – denoted by 'All' in the tables), the *percentage reduction in test suite size* and the *percentage reduction in fault detection effectiveness* metrics are calculated. The median, minimum (min), maximum (max), mean, standard deviation (stdev), 2.5^{th} percentile, and 97.5^{th} percentile values are recorded.

Table C-7 – Detailed Results for the All-Techniques-Combined Coverage.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	66.3 (117)	66.3 (117)	66.3 (117)	66.3 (117)	0 (0)	66.3 (117)	66.3 (117)
4	80.3 (118)	80.1 (119)	80.5 (117)	80.3 (118)	0 (0.14)	80.3 (118)	80.3 (118)
5	77.9 (167)	77.9 (167)	77.9 (167)	77.9 (167)	0 (0)	77.9 (167)	77.9 (167)
6	76.7 (198)	76.7 (198)	76.7 (198)	76.7 (198)	0 (0)	76.7 (198)	76.7 (198)
All	76.5 (600)	76.4 (601)	76.5 (599)	76.5 (600)	0 (0.14)	76.5 (600)	76.5 (600)

Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	20.0 (24)	13.3 (26)	23.3 (23)	19.3 (24.22)	2.0 (0.60)	16.7 (25)	23.3 (23)
4	80.0 (1)	60.0 (2)	100 (0)	77.8 (1.11)	14.5 (0.72)	60.0 (2)	100 (0)
5	0 (1)	0 (1)	0 (1)	0 (1)	0 (0)	0 (1)	0 (1)
6	0 (1)	0 (1)	0 (1)	0 (1)	0 (0)	0 (1)	0 (1)
All	27.0 (27)	18.9 (30)	32.4 (25)	26.1 (27.33)	2.7 (1.01)	21.6 (29)	29.7 (26)

Table C-8 – Detailed Results for the Function-Entry Coverage.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	96.5 (12)	96.5 (12)	96.5 (12)	96.5 (12)	0 (0)	96.5 (12)	96.5 (12)
4	97.3 (16)	97.3 (16)	97.3 (16)	97.3 (16)	0 (0)	97.3 (16)	97.3 (16)
5	97.6 (18)	97.6 (18)	97.6 (18)	97.6 (18)	0 (0)	97.6 (18)	97.6 (18)
6	97.5 (21)	97.5 (21)	97.5 (21)	97.5 (21)	0 (0)	97.5 (21)	97.5 (21)
All	97.4 (67)	97.4 (67)	97.4 (67)	97.4 (67)	0 (0)	97.4 (67)	97.4 (67)

Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	100 (0)	100 (0)	100 (0)	100 (0)	0 (0)	100 (0)	100 (0)
4	100 (0)	100 (0)	100 (0)	100 (0)	0 (0)	100 (0)	100 (0)
5	100 (0)	100 (0)	100 (0)	100 (0)	0 (0)	100 (0)	100 (0)
6	100 (0)	100 (0)	100 (0)	100 (0)	0 (0)	100 (0)	100 (0)
All	100 (0)	100 (0)	100 (0)	100 (0)	0 (0)	100 (0)	100 (0)

Table C-9 – Detailed Results for the Function-Return Coverage.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	95.4 (16)	95.4 (16)	95.4 (16)	95.4 (16)	0 (0)	95.4 (16)	95.4 (16)
4	91.7 (50)	91.7 (50)	91.7 (50)	91.7 (50)	0 (0)	91.7 (50)	91.7 (50)
5	93.1 (52)	93.1 (52)	93.1 (52)	93.1 (52)	0 (0)	93.1 (52)	93.1 (52)
6	93.4 (56)	93.4 (56)	93.4 (56)	93.4 (56)	0 (0)	93.4 (56)	93.4 (56)
All	93.2 (174)	93.2 (174)	93.2 (174)	93.2 (174)	0 (0)	93.2 (174)	93.2 (174)

Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	96.7 (1)	96.7 (1)	100 (0)	98.3 (0.52)	1.7 (0.50)	96.7 (1)	100 (0)
4	100 (0)	100 (0)	100 (0)	100 (0)	0 (0)	100 (0)	100 (0)
5	100 (0)	100 (0)	100 (0)	100 (0)	0 (0)	100 (0)	100 (0)
6	100 (0)	100 (0)	100 (0)	100 (0)	0 (0)	100 (0)	100 (0)
All	97.3 (1)	97.3 (1)	100 (0)	98.6 (0.52)	1.4 (0.50)	97.3 (1)	100 (0)

Table C-10 – Detailed Results for the Block Coverage.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	70.3 (103)	70.3 (103)	70.3 (103)	70.3 (103)	0 (0)	70.3 (103)	70.3 (103)
4	82.1 (107)	82.1 (107)	82.3 (106)	82.1 (106.99)	0 (0.1)	82.1 (107)	82.1 (107)
5	81.0 (143)	81.0 (143)	81.0 (143)	81.0 (143)	0 (0)	81.0 (143)	81.0 (143)
6	80.1 (169)	80.1 (169)	80.1 (169)	80.1 (169)	0 (0)	80.1 (169)	80.1 (169)
All	79.5 (522)	79.5 (522)	79.6 (521)	79.5 (521.99)	0 (0.1)	79.5 (522)	79.5 (522)

Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	23.3 (23)	20.0 (24)	26.7 (22)	22.7 (23.19)	1.9 (0.56)	20.0 (24)	26.7 (22)
4	80.0 (1)	60.0 (2)	100 (0)	77.8 (1.11)	14.5 (0.72)	60.0 (2)	100 (0)
5	0 (1)	0 (1)	0 (1)	0 (1)	0 (0)	0 (1)	0 (1)
6	0 (1)	0 (1)	0 (1)	0 (1)	0 (0)	0 (1)	0 (1)
All	29.7 (26)	24.3 (28)	35.1 (24)	28.9 (26.3)	2.6 (0.98)	24.3 (28)	32.4 (25)

Table C-11 – Detailed Results for the Basic-Block Coverage.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	70.3 (103)	70.3 (103)	70.3 (103)	70.3 (103)	0 (0)	70.3 (103)	70.3 (103)
4	82.1 (107)	82.1 (107)	82.3 (106)	82.1 (106.99)	0 (0.1)	82.1 (107)	82.1 (107)
5	81.0 (143)	81.0 (143)	81.0 (143)	81.0 (143)	0 (0)	81.0 (143)	81.0 (143)
6	80.1 (169)	80.1 (169)	80.1 (169)	80.1 (169)	0 (0)	80.1 (169)	80.1 (169)
All	79.5 (522)	79.5 (522)	79.6 (521)	79.5 (521.99)	0 (0.1)	79.5 (522)	79.5 (522)

Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	23.3 (23)	20.0 (24)	26.7 (22)	22.7 (23.19)	1.9 (0.56)	20.0 (24)	26.7 (22)
4	80.0 (1)	60.0 (2)	100 (0)	77.8 (1.11)	14.5 (0.72)	60.0 (2)	100 (0)
5	0 (1)	0 (1)	0 (1)	0 (1)	0 (0)	0 (1)	0 (1)
6	0 (1)	0 (1)	0 (1)	0 (1)	0 (0)	0 (1)	0 (1)
All	29.7 (26)	24.3 (28)	35.1 (24)	28.9 (26.3)	2.6 (0.98)	24.3 (28)	32.4 (25)

Table C-12 – Detailed Results for the Decision Coverage.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	72.0 (97)	72.0 (97)	72.0 (97)	72.0 (97)	0 (0)	72.0 (97)	72.0 (97)
4	81.8 (109)	81.6 (110)	82.0 (108)	81.8 (109)	0 (0.14)	81.8 (109)	81.8 (109)
5	79.3 (156)	79.3 (156)	79.3 (156)	79.3 (156)	0 (0)	79.3 (156)	79.3 (156)
6	78.2 (185)	78.2 (185)	78.2 (185)	78.2 (185)	0 (0)	78.2 (185)	78.2 (185)
All	78.5 (547)	78.5 (548)	78.6 (546)	78.5 (547)	0 (0.14)	78.5 (547)	78.5 (547)

Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	20.0 (24)	13.3 (26)	23.3 (23)	20.3 (23.91)	2.4 (0.71)	16.7 (25)	23.3 (23)
4	60.0 (2)	60.0 (2)	80.0 (1)	65.8 (1.71)	9.1 (0.46)	60.0 (2)	80.0 (1)
5	0 (1)	0 (1)	0 (1)	0 (1)	0 (0)	0 (1)	0 (1)
6	0 (1)	0 (1)	0 (1)	0 (1)	0 (0)	0 (1)	0 (1)
All	24.3 (28)	18.9 (30)	29.7 (26)	25.4 (27.62)	2.4 (0.90)	21.6 (29)	29.7 (26)

C.3 Prioritization

The following tables contain the detailed results of the prioritization executions for each of the techniques, except M1 and M3, described in Table 2-8. As explained in section 3.7.5, each of these prioritization techniques is executed against 100 randomly ordered trace files for each of the software versions. For each of the techniques, for each software version, and for the weighted average (see section 3.7.4) of the software versions (denoted by 'All' in the tables), the median, minimum (min), maximum (max), mean, standard deviation (stdev), 2.5th percentile, and 97.5th percentile APFD scores are presented.

The untreated (M1) and optimal (M3) techniques are not listed here because unlike the other techniques, both of these control techniques require only that a single ordering be considered. The untreated technique is simply the test suite's original ordering for each of the software versions. The optimal technique is the most favourable ordering that maximizes the APFD value. These values are recorded in Table 4-9.

Table C-13 – Detailed Results for the Random (M2) Technique.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5th percentile)	Upper Bound (97.5th percentile)
3	71.4	62.0	78.2	70.9	4.0	62.8	77.8
4	60.7	36.8	95.6	61.8	13.6	39.5	87.5
5	94.0	64.5	99.9	91.3	7.8	69.5	99.7
6	93.9	67.1	99.9	92.2	7.2	72.9	99.8
All	70.6	61.5	80.5	70.8	3.6	64.0	77.2

Table C-14 – Detailed Results for the Function-Entry-Total (M4) Technique.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5th percentile)	Upper Bound (97.5th percentile)
3	56.3	55.7	57.0	56.3	0.3	55.8	56.8
4	67.0	66.5	67.2	67.0	0.1	66.7	67.2
5	50.1	48.1	50.7	50.0	0.6	48.8	50.7
6	71.7	71.7	71.7	71.7	0.0	71.7	71.7
All	58.0	57.5	58.5	58.0	0.2	57.5	58.4

Table C-15 – Detailed Results for the Function-Return-Total (M5) Technique.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	55.9	55.3	56.5	55.9	0.3	55.3	56.4
4	68.5	67.9	69.1	68.5	0.2	68.0	68.9
5	48.9	46.6	49.5	48.7	0.7	47.3	49.5
6	71.9	71.8	71.9	71.9	0.1	71.8	71.9
All	57.8	57.3	58.3	57.8	0.2	57.4	58.2

Table C-16 – Detailed Results for the Block-Total (M6) Technique.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	57.9	57.8	58.0	57.9	0.0	57.8	58.0
4	70.6	70.6	70.6	70.6	0.0	70.6	70.6
5	55.4	55.0	55.6	55.4	0.2	55.0	55.6
6	64.5	64.5	64.5	64.5	0.0	64.5	64.5
All	59.7	59.7	59.8	59.7	0.0	59.7	59.8

Table C-17 – Detailed Results for the Basic-Block-Total (M7) Technique.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	59.6	59.5	59.7	59.6	0.0	59.5	59.7
4	71.0	70.9	71.0	71.0	0.0	70.9	71.0
5	55.4	55.0	55.5	55.3	0.2	55.0	55.5
6	65.9	65.9	65.9	65.9	0.0	65.9	65.9
All	61.2	61.1	61.2	61.2	0.0	61.1	61.2

Table C-18 – Detailed Results for the Decision-Total (M8) Technique.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	55.6	55.5	55.8	55.6	0.1	55.5	55.8
4	65.7	65.5	65.9	65.7	0.1	65.5	65.8
5	59.9	59.1	60.3	59.8	0.4	59.1	60.3
6	67.2	67.1	67.3	67.2	0.1	67.1	67.3
All	57.4	57.3	57.6	57.4	0.1	57.3	57.5

Table C-19 – Detailed Results for the Function-Entry-Addtl (M9) Technique.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	57.8	54.0	60.3	57.7	1.3	55.0	59.7
4	71.9	67.5	78.3	72.1	2.3	68.5	76.7
5	84.2	67.4	95.6	82.9	7.5	69.7	94.5
6	77.6	70.4	87.0	79.7	5.2	71.6	86.8
All	61.1	57.7	63.5	60.9	1.2	58.5	62.8

Table C-20 – Detailed Results for the Function-Return-Addtl (M10) Technique.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	56.6	53.9	59.1	56.7	1.2	54.5	58.8
4	72.7	67.2	82.3	72.7	3.0	68.1	78.4
5	62.9	50.3	90.6	64.8	8.6	53.6	84.3
6	70.1	64.4	73.9	69.0	1.9	65.0	71.4
All	59.5	56.7	62.5	59.4	1.1	57.4	61.3

Table C-21 – Detailed Results for the Block-Addtl (M11) Technique.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	74.3	73.1	76.6	74.4	0.7	73.5	76.2
4	67.6	58.2	77.0	67.4	3.9	60.5	75.9
5	91.3	87.1	95.4	91.2	1.5	87.3	95.3
6	84.2	81.7	85.0	84.1	0.6	82.7	85.0
All	74.1	72.3	75.9	74.2	0.7	72.8	75.8

Table C-22 – Detailed Results for the Basic-Block-Addtl (M12) Technique.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	78.0	76.6	79.8	78.0	0.7	76.9	79.7
4	67.0	56.8	81.0	67.0	4.5	59.4	75.5
5	89.3	85.1	94.6	90.1	2.5	86.7	94.6
6	89.8	88.8	89.9	89.8	0.2	89.3	89.9
All	77.2	75.4	79.6	77.2	0.9	75.6	79.2

Table C-23 – Detailed Results for the Decision-Addtl (M13) Technique.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	74.1	72.4	77.3	74.3	1.0	72.7	76.5
4	71.1	60.3	75.1	69.9	3.6	61.7	75.0
5	93.4	92.2	95.8	93.3	0.5	92.5	94.0
6	96.8	96.2	96.8	96.7	0.1	96.4	96.8
All	74.7	72.8	77.6	74.8	1.0	73.0	76.8

Table C-24 – Detailed Results for the Function-Entry-Diff-Total (M14) Technique.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	55.6	54.8	56.3	55.6	0.3	55.0	56.2
4	65.0	64.7	65.3	65.0	0.1	64.8	65.2
5	53.0	51.9	53.2	52.9	0.3	52.3	53.2
6	70.8	70.1	71.6	70.8	0.5	70.1	71.6
All	57.2	56.6	57.8	57.2	0.3	56.7	57.7

Table C-25 – Detailed Results for the Function-Return-Diff-Total (M15) Technique.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	55.6	54.9	56.3	55.6	0.3	55.0	56.1
4	65.6	65.1	65.9	65.6	0.2	65.2	65.8
5	46.4	43.0	47.3	46.2	1.0	43.9	47.3
6	76.9	75.9	78.0	76.9	0.7	75.9	78.0
All	57.2	56.7	57.8	57.2	0.2	56.8	57.7

Table C-26 – Detailed Results for the Block-Diff-Total (M16) Technique.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	58.3	58.1	58.4	58.3	0.0	58.2	58.4
4	65.0	65.0	65.0	65.0	0.0	65.0	65.0
5	61.1	60.7	61.5	61.1	0.3	60.7	61.5
6	72.4	72.3	72.5	72.4	0.1	72.3	72.5
All	59.6	59.5	59.7	59.6	0.0	59.5	59.7

Table C-27 – Detailed Results for the Basic-Block-Diff-Total (M17) Technique.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	59.7	59.6	59.9	59.7	0.0	59.7	59.8
4	66.9	66.8	66.9	66.9	0.0	66.8	66.9
5	64.0	63.9	64.0	63.9	0.1	63.9	64.0
6	74.6	74.6	74.6	74.6	0.0	74.6	74.6
All	61.2	61.1	61.3	61.2	0.0	61.2	61.3

Table C-28 – Detailed Results for the Decision-Diff-Total (M18) Technique.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	55.7	55.5	55.8	55.7	0.1	55.6	55.8
4	64.2	64.1	64.3	64.2	0.1	64.1	64.3
5	64.7	64.5	64.7	64.6	0.1	64.5	64.7
6	68.0	67.8	68.1	68.0	0.1	67.8	68.1
All	57.4	57.3	57.5	57.4	0.1	57.3	57.5

Table C-29 – Detailed Results for the Function-Entry-Diff-Addtl (M19) Technique.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	57.7	54.1	60.2	57.6	1.4	54.8	59.6
4	72.4	67.5	80.5	72.5	2.4	68.7	77.4
5	69.8	50.2	83.6	67.2	7.7	51.4	79.3
6	70.7	63.9	75.2	70.8	2.2	65.8	74.9
All	60.3	56.9	62.3	60.2	1.2	57.9	61.9

Table C-30 – Detailed Results for the Function-Return-Diff-Addtl (M20) Technique.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	60.8	57.9	63.5	60.9	1.2	58.1	62.9
4	63.1	57.0	71.3	63.3	3.4	57.3	69.6
5	67.6	55.2	80.2	67.9	5.4	56.9	75.8
6	59.7	53.5	80.4	60.4	3.9	55.0	69.6
All	61.4	58.9	63.6	61.4	1.0	59.2	63.2

Table C-31 – Detailed Results for the Block-Diff-Addtl (M21) Technique.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	74.8	73.9	77.3	75.0	0.7	74.1	76.7
4	60.3	55.2	68.1	60.4	2.8	56.2	65.6
5	93.2	89.9	93.4	93.0	0.7	90.6	93.4
6	99.2	86.7	99.2	96.2	4.9	87.4	99.2
All	74.0	72.6	76.2	74.1	0.7	72.9	75.6

Table C-32 – Detailed Results for the Basic-Block-Diff-Addtl (M22) Technique.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	78.3	77.2	80.7	78.4	0.7	77.4	80.4
4	62.1	56.4	68.9	62.2	2.9	57.1	68.0
5	92.1	86.1	93.2	91.8	1.4	87.6	93.2
6	92.8	91.6	93.0	92.7	0.2	92.2	93.0
All	76.9	75.7	78.8	77.0	0.7	76.0	78.4

Table C-33 – Detailed Results for the Decision-Diff-Addtl (M23) Technique.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	76.3	75.0	79.0	76.4	0.8	75.3	78.5
4	67.0	59.7	74.5	67.2	3.3	61.4	73.8
5	96.4	95.8	96.5	96.3	0.2	96.0	96.5
6	99.6	99.6	99.6	99.6	0.0	99.6	99.6
All	76.2	74.6	78.6	76.4	0.8	75.1	78.3

Appendix D Detailed Experimental Results Considering Only Versions 3 and 4 of NoiseGen

The following subsections detail the experimental results considering only versions 3 and 4 of the NoiseGen software. Averages using only these results are also calculated. This is done for comparison to the results of section 4 and Appendix C where all the NoiseGen versions are considered. Versions 5 and 6 of NoiseGen contain only one fault each and it was a concern that their inclusion in the calculation of the average of the data might skew the results.

The resulting tables and figures in the following sections closely resemble those that were generated using the experimental data from all of the NoiseGen versions.

D.1 Retest-all

The following table summarizes the results of the retest-all control technique for software versions 3 and 4, and for software versions 3 and 4 combined (denoted by '3 + 4' in the tables). Table D-1 summarizes the resulting savings in absolute terms and using the *percentage reduction in test suite size* metric and it also summarizes the resulting costs in absolute terms and using the *percentage reduction in fault detection effectiveness* metric.

Table D-1 – Results for the Retest-all Technique Considering Versions 3 and 4 Only.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)
3	0 (347)
4	0 (599)
3 + 4	0 (946)
Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)
3	0 (30)
4	0 (5)
3 + 4	0 (35)

D.2 Modification-Based Regression Test Selection

The following table summarizes the results of the modification-based regression test selection for each of the code coverage types used, for software versions 3 and 4, and for software versions 3 and 4 combined (to get average values across these versions – denoted by '3 + 4' in the tables). Table D-2 summarizes the resulting savings in absolute terms and using the *percentage reduction in test suite size* metric and it also summarizes the resulting costs in absolute terms and using the *percentage reduction in fault detection effectiveness* metric.

Table D-2 – Results for the Modification-Based Regression Test Selection Strategy Considering Versions 3 and 4 Only.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)					
	All Techniques Combined	Function Entry	Function Return	Block	Basic Block	Decision
3	0 (347)	0 (347)	0 (347)	0 (347)	0 (347)	0 (347)
4	0 (599)	1.2 (592)	0.2 (598)	0 (599)	0 (599)	0 (599)
3 + 4	0 (946)	0.7 (939)	0.1 (945)	0 (946)	0 (946)	0 (946)
Software Version	**Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)**					
	All Techniques Combined	Function Entry	Function Return	Block	Basic Block	Decision
3	0 (30)	0 (30)	0 (30)	0 (30)	0 (30)	0 (30)
4	0 (5)	0 (5)	0 (5)	0 (5)	0 (5)	0 (5)
3 + 4	0 (35)	0 (35)	0 (35)	0 (35)	0 (35)	0 (35)

D.3 Modification-Focused Minimization

The following tables contain the detailed results of the modification-focused minimization executions for each coverage criterion used. As explained in section 3.7.5, each of the minimizations is executed against 100 randomly ordered trace files for each of the software versions. For each coverage criterion, for software versions 3 and 4, and software versions 3 and 4 combined (to get average values across these versions – denoted by '3 + 4' in the tables), the *percentage reduction in test suite size* and the

percentage reduction in fault detection effectiveness metrics are calculated. The median, minimum (min), maximum (max), mean, standard deviation (stdev), 2.5th percentile, and 97.5th percentile values are recorded.

Figure D-1 presents the test set size savings across versions 3 and 4 of all the modification-focused minimizations as a set of boxplots. Figure D-2 presents the reduction in fault detection effectiveness across versions 3 and 4 of all the modification-focused minimizations as a set of boxplots.

Table D-3 – Detailed Results for the All-Techniques-Combined Coverage Considering Versions 3 and 4 Only.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5th percentile)	Upper Bound (97.5th percentile)
3	70.9 (101)	70.9 (101)	70.9 (101)	70.9 (101)	0 (0)	70.9 (101)	70.9 (101)
4	87.8 (73)	87.8 (73)	87.8 (73)	87.8 (73)	0 (0)	87.8 (73)	87.8 (73)
3 + 4	81.6 (174)	81.6 (174)	81.6 (174)	81.6 (174)	0 (0)	81.6 (174)	81.6 (174)
Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5th percentile)	Upper Bound (97.5th percentile)
3	20.0 (24)	16.7 (25)	23.3 (23)	19.4 (24.19)	1.9 (0.56)	16.7 (25)	23.3 (23)
4	80.0 (1)	60.0 (2)	100 (0)	82.0 (0.9)	12.9 (0.64)	60.0 (2)	100 (0)
3 + 4	28.6 (25)	22.9 (27)	34.3 (23)	28.3 (25.09)	2.6 (0.92)	22.9 (27)	32.9 (23.48)

Table D-4 – Detailed Results for the Function-Entry Coverage Considering Versions 3 and 4 Only.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	96.5 (12)	96.5 (12)	96.5 (12)	96.5 (12)	0 (0)	96.5 (12)	96.5 (12)
4	97.3 (16)	97.3 (16)	97.3 (16)	97.3 (16)	0 (0)	97.3 (16)	97.3 (16)
3 + 4	97.0 (28)	97.0 (28)	97.0 (28)	97.0 (28)	0 (0)	97.0 (28)	97.0 (28)

Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	100 (0)	100 (0)	100 (0)	100 (0)	0 (0)	100 (0)	100 (0)
4	100 (0)	100 (0)	100 (0)	100 (0)	0 (0)	100 (0)	100 (0)
3 + 4	100 (0)	100 (0)	100 (0)	100 (0)	0 (0)	100 (0)	100 (0)

Table D-5 – Detailed Results for the Function-Return Coverage Considering Versions 3 and 4 Only.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	96.8 (11)	96.8 (11)	96.8 (11)	96.8 (11)	0 (0)	96.8 (11)	96.8 (11)
4	96.2 (23)	96.2 (23)	96.2 (23)	96.2 (23)	0 (0)	96.2 (23)	96.2 (23)
3 + 4	96.4 (34)	96.4 (34)	96.4 (34)	96.4 (34)	0 (0)	96.4 (34)	96.4 (34)

Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	100 (0)	96.7 (1)	100 (0)	98.4 (0.47)	1.7 (0.50)	96.7 (1)	100 (0)
4	100 (0)	80.0 (1)	100 (0)	91.2 (0.44)	10.0 (0.50)	80.0 (1)	100 (0)
3 + 4	97.1 (1)	94.3 (2)	100 (0)	97.4 (0.91)	2.2 (0.77)	94.3 (2)	100 (0)

Table D-6 – Detailed Results for the Block Coverage Considering Versions 3 and 4 Only.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	72.9 (94)	72.9 (94)	72.9 (94)	72.9 (94)	0 (0)	72.9 (94)	72.9 (94)
4	89.1 (65)	89.1 (65)	89.1 (65)	89.1 (65)	0 (0)	89.1 (65)	89.1 (65)
3 + 4	83.2 (159)	83.2 (159)	83.2 (159)	83.2 (159)	0 (0)	83.2 (159)	83.2 (159)

Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	23.3 (23)	20.0 (24)	26.7 (22)	22.7 (23.19)	1.9 (0.56)	20.0 (24)	26.7 (22)
4	80.0 (1)	80.0 (1)	100 (0)	85.8 (0.71)	9.1 (0.46)	80.0 (1)	100 (0)
3 + 4	31.4 (24)	28.6 (25)	37.1 (22)	31.7 (23.9)	2.3 (0.81)	28.6 (25)	37.1 (22)

Table D-7 – Detailed Results for the Basic-Block Coverage Considering Versions 3 and 4 Only.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	72.9 (94)	72.9 (94)	72.9 (94)	72.9 (94)	0 (0)	72.9 (94)	72.9 (94)
4	89.1 (65)	89.1 (65)	89.1 (65)	89.1 (65)	0 (0)	89.1 (65)	89.1 (65)
3 + 4	83.2 (159)	83.2 (159)	83.2 (159)	83.2 (159)	0 (0)	83.2 (159)	83.2 (159)

Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	23.3 (23)	20.0 (24)	26.7 (22)	22.7 (23.19)	1.9 (0.56)	20.0 (24)	26.7 (22)
4	80.0 (1)	80.0 (1)	100 (0)	85.8 (0.71)	9.1 (0.46)	80.0 (1)	100 (0)
3 + 4	31.4 (24)	28.6 (25)	37.1 (22)	31.7 (23.9)	2.3 (0.81)	28.6 (25)	37.1 (22)

Table D-8 – Detailed Results for the Decision Coverage Considering Versions 3 and 4 Only.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	76.1 (83)	76.1 (83)	76.1 (83)	76.1 (83)	0 (0)	76.1 (83)	76.1 (83)
4	88.8 (67)	88.8 (67)	88.8 (67)	88.8 (67)	0 (0)	88.8 (67)	88.8 (67)
3 + 4	84.1 (150)	84.1 (150)	84.1 (150)	84.1 (150)	0 (0)	84.1 (150)	84.1 (150)
Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	20.0 (24)	16.7 (25)	23.3 (23)	20.4 (23.88)	2.3 (0.69)	16.7 (25)	23.3 (23)
4	80.0 (1)	60.0 (2)	100 (0)	82.0 (0.9)	12.9 (0.64)	60.0 (2)	100 (0)
3 + 4	28.6 (25)	22.9 (27)	34.3 (23)	29.2 (24.78)	2.9 (1.0)	22.9 (27)	34.3 (23)

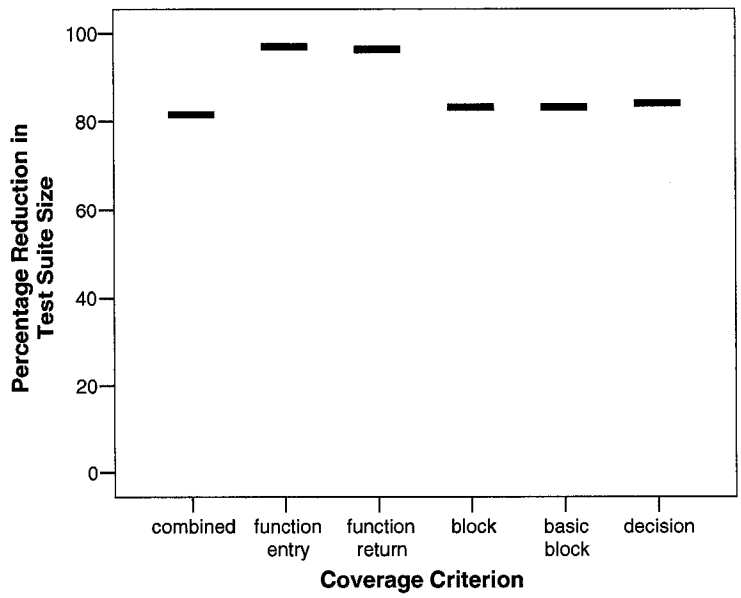


Figure D-1 – Average test set size savings for the modification-focused minimizations across versions 3 and 4.

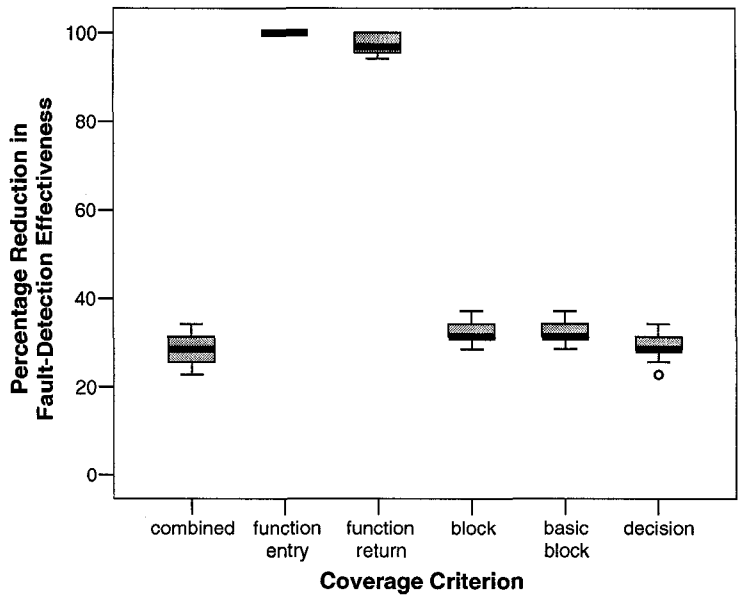


Figure D-2 – Average reduction in fault detection effectiveness for the modification-focused minimizations across versions 3 and 4.

D.4 Coverage-Focused Minimization

The following tables contain the detailed results of the coverage-focused minimization executions for each coverage criterion used. As explained in section 3.7.5, each of the test suite reductions is executed against 100 randomly ordered trace files for each of the software versions. For each coverage criterion, for software versions 3 and 4, and software versions 3 and 4 combined (to get average values across these versions – denoted by '3 + 4' in the tables), the *percentage reduction in test suite size* and the *percentage reduction in fault detection effectiveness* metrics are calculated. The median, minimum (min), maximum (max), mean, standard deviation (stdev), 2.5^{th} percentile, and 97.5^{th} percentile values are recorded.

Figure D-3 presents the test set size savings across versions 3 and 4 of all the coverage-focused minimizations as a set of boxplots. Figure D-4 presents the reduction in fault detection effectiveness across versions 3 and 4 of all the coverage-focused minimizations as a set of boxplots.

Table D-9 – Detailed Results for the All-Techniques-Combined Coverage Considering Versions 3 and 4 Only.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	66.3 (117)	66.3 (117)	66.3 (117)	66.3 (117)	0 (0)	66.3 (117)	66.3 (117)
4	80.3 (118)	80.1 (119)	80.5 (117)	80.3 (118)	0 (0.14)	80.3 (118)	80.3 (118)
3 + 4	75.2 (235)	75.1 (236)	75.3 (234)	75.2 (235)	0 (0.14)	75.2 (235)	75.2 (235)
Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	20.0 (24)	13.3 (26)	23.3 (23)	19.3 (24.22)	2.0 (0.60)	16.7 (25)	23.3 (23)
4	80.0 (1)	60.0 (2)	100 (0)	77.8 (1.11)	14.5 (0.72)	60.0 (2)	100 (0)
3 + 4	28.6 (25)	20.0 (28)	34.3 (23)	27.6 (25.33)	2.9 (1.01)	22.9 (27)	31.4 (24)

Table D-10 – Detailed Results for the Function-Entry Coverage Considering Versions 3 and 4 Only.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	96.5 (12)	96.5 (12)	96.5 (12)	96.5 (12)	0 (0)	96.5 (12)	96.5 (12)
4	97.3 (16)	97.3 (16)	97.3 (16)	97.3 (16)	0 (0)	97.3 (16)	97.3 (16)
3 + 4	97.0 (28)	97.0 (28)	97.0 (28)	97.0 (28)	0 (0)	97.0 (28)	97.0 (28)
Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	100 (0)	100 (0)	100 (0)	100 (0)	0 (0)	100 (0)	100 (0)
4	100 (0)	100 (0)	100 (0)	100 (0)	0 (0)	100 (0)	100 (0)
3 + 4	100 (0)	100 (0)	100 (0)	100 (0)	0 (0)	100 (0)	100 (0)

Table D-11 – Detailed Results for the Function-Return Coverage Considering Versions 3 and 4 Only.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	95.4 (16)	95.4 (16)	95.4 (16)	95.4 (16)	0 (0)	95.4 (16)	95.4 (16)
4	91.7 (50)	91.7 (50)	91.7 (50)	91.7 (50)	0 (0)	91.7 (50)	91.7 (50)
3 + 4	93.0 (66)	93.0 (66)	93.0 (66)	93.0 (66)	0 (0)	93.0 (66)	93.0 (66)

Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	96.7 (1)	96.7 (1)	100 (0)	98.3 (0.52)	1.7 (0.50)	96.7 (1)	100 (0)
4	100 (0)	100 (0)	100 (0)	100 (0)	0 (0)	100 (0)	100 (0)
3 + 4	97.1 (1)	97.1 (1)	100 (0)	98.5 (0.52)	1.4 (0.50)	97.1 (1)	100 (0)

Table D-12 – Detailed Results for the Block Coverage Considering Versions 3 and 4 Only.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	70.3 (103)	70.3 (103)	70.3 (103)	70.3 (103)	0 (0)	70.3 (103)	70.3 (103)
4	82.1 (107)	82.1 (107)	82.3 (106)	82.1 (106.99)	0 (0.1)	82.1 (107)	82.1 (107)
3 + 4	77.8 (210)	77.8 (210)	77.9 (209)	77.8 (209.99)	0 (0.1)	77.8 (210)	77.8 (210)

Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	23.3 (23)	20.0 (24)	26.7 (22)	22.7 (23.19)	1.9 (0.56)	20.0 (24)	26.7 (22)
4	80.0 (1)	60.0 (2)	100 (0)	77.8 (1.11)	14.5 (0.72)	60.0 (2)	100 (0)
3 + 4	31.4 (24)	25.7 (26)	37.1 (22)	30.6 (24.3)	2.8 (0.98)	25.7 (26)	34.3 (23)

Table D-13 – Detailed Results for the Basic-Block Coverage Considering Versions 3 and 4 Only.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	70.3 (103)	70.3 (103)	70.3 (103)	70.3 (103)	0 (0)	70.3 (103)	70.3 (103)
4	82.1 (107)	82.1 (107)	82.3 (106)	82.1 (106.99)	0 (0.1)	82.1 (107)	82.1 (107)
3 + 4	77.8 (210)	77.8 (210)	77.9 (209)	77.8 (209.99)	0 (0.1)	77.8 (210)	77.8 (210)

Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	23.3 (23)	20.0 (24)	26.7 (22)	22.7 (23.19)	1.9 (0.56)	20.0 (24)	26.7 (22)
4	80.0 (1)	60.0 (2)	100 (0)	77.8 (1.11)	14.5 (0.72)	60.0 (2)	100 (0)
3 + 4	31.4 (24)	25.7 (26)	37.1 (22)	30.6 (24.3)	2.8 (0.98)	25.7 (26)	34.3 (23)

Table D-14 – Detailed Results for the Decision Coverage Considering Versions 3 and 4 Only.

Software Version	Percentage Reduction in Test Suite Size (Number of test cases selected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	72.0 (97)	72.0 (97)	72.0 (97)	72.0 (97)	0 (0)	72.0 (97)	72.0 (97)
4	81.8 (109)	81.6 (110)	82.0 (108)	81.8 (109)	0 (0.14)	81.8 (109)	81.8 (109)
3 + 4	78.2 (206)	78.1 (207)	78.3 (205)	78.2 (206)	0 (0.14)	78.2 (206)	78.2 (206)

Software Version	Percentage Reduction in Fault Detection Effectiveness (Number of faults detected)						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	20.0 (24)	13.3 (26)	23.3 (23)	20.3 (23.91)	2.4 (0.71)	16.7 (25)	23.3 (23)
4	60.0 (2)	60.0 (2)	80.0 (1)	65.8 (1.71)	9.1 (0.46)	60.0 (2)	80.0 (1)
3 + 4	25.7 (26)	20.0 (28)	31.4 (24)	26.8 (25.62)	2.6 (0.90)	22.9 (27)	31.4 (24)

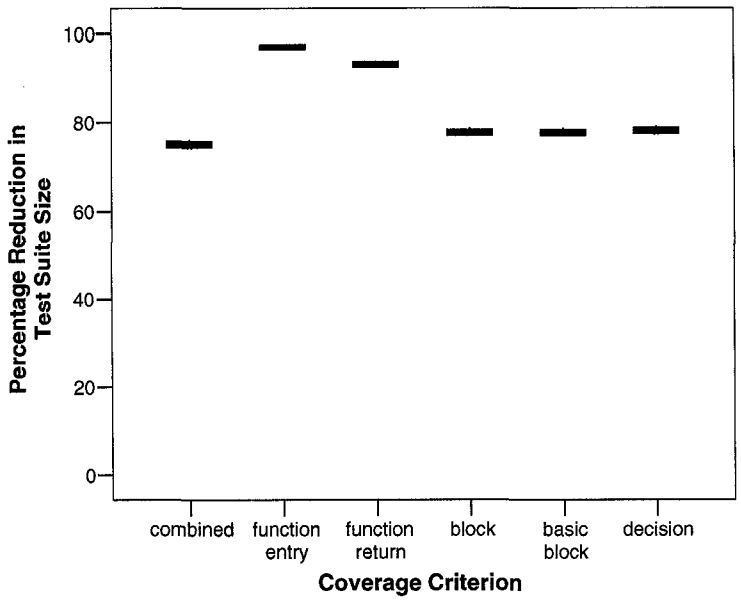


Figure D-3 – Average test set size savings for the coverage-focused minimizations across versions 3 and 4.

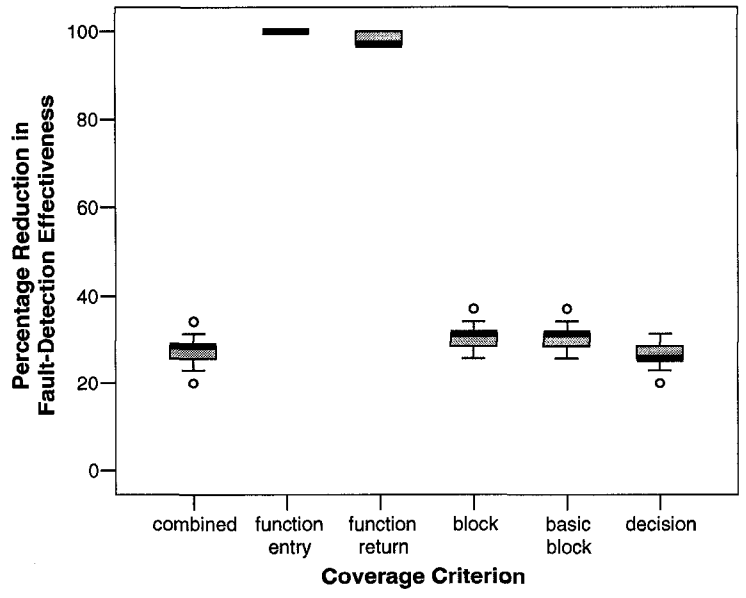


Figure D-4 – Average reduction in fault detection effectiveness for the coverage-focused minimizations across versions 3 and 4.

D.5 Prioritization

The following tables contain the detailed results of the prioritization executions for each of the techniques described in Table 2-8. With the exception of techniques M1 and M3, as explained in section 3.7.5, each of the prioritization techniques is executed against 100 randomly ordered trace files for each of the software versions. For each of these techniques, for software versions 3 and 4, and for the weighted average (see below) of software versions 3 and 4 (denoted by '3 + 4' in the tables), the median, minimum (min), maximum (max), mean, standard deviation (stdev), 2.5th percentile, and 97.5th percentile APFD scores are presented.

The weighted average APFD value is calculated for software versions 3 and 4 combined for all the techniques. This is a weighted average that gives more significance to version 3's test suite that detects more faults. Given that 35 faults are detected across the two software versions considered (30 and 5 faults in versions 3 and 4 respectively), then

$$APFD_{ver.3+4} = \frac{30}{35} \times APFD_{ver.3} + \frac{5}{35} \times APFD_{ver.4}$$

Figure D-5 presents the results of all the prioritizations as a set of boxplots. The boxplots represent the $APFD_{ver.3+4}$ values for each of the prioritization techniques studied. The prioritization techniques are described in Table 2-8.

Table D-15 – Results for the Untreated (M1) and Optimal (M3) Techniques Considering Versions 3 and 4 Only.

Software Version	APFD Scores per Technique Used	
	M1	M3
3	56.9	97.1
4	23.4	99.7
3 + 4	52.1	97.5

Table D-16 – Detailed Results for the Random (M2) Technique Considering Versions 3 and 4 Only.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	71.4	62.0	78.2	70.9	4.0	62.8	77.8
4	60.7	36.8	95.6	61.8	13.6	39.5	87.5
3 + 4	69.6	60.0	79.9	69.6	3.8	62.8	76.4

Table D-17 – Detailed Results for the Function-Entry-Total (M4) Technique Considering Versions 3 and 4 Only.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	56.3	55.7	57.0	56.3	0.3	55.8	56.8
4	67.0	66.5	67.2	67.0	0.1	66.7	67.2
3 + 4	57.8	57.3	58.4	57.8	0.2	57.3	58.3

Table D-18 – Detailed Results for the Function-Return-Total (M5) Technique Considering Versions 3 and 4 Only.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	55.9	55.3	56.5	55.9	0.3	55.3	56.4
4	68.5	67.9	69.1	68.5	0.2	68.0	68.9
3 + 4	57.7	57.2	58.2	57.7	0.2	57.2	58.1

Table D-19 – Detailed Results for the Block-Total (M6) Technique Considering Versions 3 and 4 Only.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	57.9	57.8	58.0	57.9	0.0	57.8	58.0
4	70.6	70.6	70.6	70.6	0.0	70.6	70.6
3 + 4	59.7	59.6	59.8	59.7	0.0	59.7	59.8

Table D-20 – Detailed Results for the Basic-Block-Total (M7) Technique Considering Versions 3 and 4 Only.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	59.6	59.5	59.7	59.6	0.0	59.5	59.7
4	71.0	70.9	71.0	71.0	0.0	70.9	71.0
3 + 4	61.2	61.1	61.3	61.2	0.0	61.1	61.3

Table D-21 – Detailed Results for the Decision-Total (M8) Technique Considering Versions 3 and 4 Only.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	55.6	55.5	55.8	55.6	0.1	55.5	55.8
4	65.7	65.5	65.9	65.7	0.1	65.5	65.8
3 + 4	57.1	56.9	57.2	57.1	0.1	57.0	57.2

Table D-22 – Detailed Results for the Function-Entry-Addtl (M9) Technique Considering Versions 3 and 4 Only.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	57.8	54.0	60.3	57.7	1.3	55.0	59.7
4	71.9	67.5	78.3	72.1	2.3	68.5	76.7
3 + 4	59.9	56.5	62.6	59.8	1.2	57.4	61.6

Table D-23 – Detailed Results for the Function-Return-Addtl (M10) Technique Considering Versions 3 and 4 Only.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	56.6	53.9	59.1	56.7	1.2	54.5	58.8
4	72.7	67.2	82.3	72.7	3.0	68.1	78.4
3 + 4	59.1	56.3	61.6	59.0	1.1	56.8	61.0

Table D-24 – Detailed Results for the Block-Addtl (M11) Technique Considering Versions 3 and 4 Only.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	74.3	73.1	76.6	74.4	0.7	73.5	76.2
4	67.6	58.2	77.0	67.4	3.9	60.5	75.9
3 + 4	73.3	71.4	75.2	73.4	0.8	72.0	75.1

Table D-25 – Detailed Results for the Basic-Block-Addtl (M12) Technique Considering Versions 3 and 4 Only.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	78.0	76.6	79.8	78.0	0.7	76.9	79.7
4	67.0	56.8	81.0	67.0	4.5	59.4	75.5
3 + 4	76.5	74.6	79.1	76.4	0.9	74.8	78.5

Table D-26 – Detailed Results for the Decision-Addtl (M13) Technique Considering Versions 3 and 4 Only.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	74.1	72.4	77.3	74.3	1.0	72.7	76.5
4	71.1	60.3	75.1	69.9	3.6	61.7	75.0
3 + 4	73.6	71.5	76.6	73.7	1.0	71.8	75.7

Table D-27 – Detailed Results for the Function-Entry-Diff-Total (M14) Technique Considering Versions 3 and 4 Only.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	55.6	54.8	56.3	55.6	0.3	55.0	56.2
4	65.0	64.7	65.3	65.0	0.1	64.8	65.2
3 + 4	56.9	56.3	57.6	56.9	0.3	56.4	57.4

Table D-28 – Detailed Results for the Function-Return-Diff-Total (M15) Technique Considering Versions 3 and 4 Only.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	55.6	54.9	56.3	55.6	0.3	55.0	56.1
4	65.6	65.1	65.9	65.6	0.2	65.2	65.8
3 + 4	57.0	56.5	57.6	57.0	0.3	56.5	57.5

Table D-29 – Detailed Results for the Block-Diff-Total (M16) Technique Considering Versions 3 and 4 Only.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	58.3	58.1	58.4	58.3	0.0	58.2	58.4
4	65.0	65.0	65.0	65.0	0.0	65.0	65.0
3 + 4	59.2	59.1	59.3	59.2	0.0	59.1	59.3

Table D-30 – Detailed Results for the Basic-Block-Diff-Total (M17) Technique Considering Versions 3 and 4 Only.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	59.7	59.6	59.9	59.7	0.0	59.7	59.8
4	66.9	66.8	66.9	66.9	0.0	66.8	66.9
3 + 4	60.8	60.7	60.9	60.8	0.0	60.7	60.8

Table D-31 – Detailed Results for the Decision-Diff-Total (M18) Technique Considering Versions 3 and 4 Only.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	55.7	55.5	55.8	55.7	0.1	55.6	55.8
4	64.2	64.1	64.3	64.2	0.1	64.1	64.3
3 + 4	56.9	56.7	57.0	56.9	0.1	56.8	57.0

Table D-32 – Detailed Results for the Function-Entry-Diff-Addtl (M19) Technique Considering Versions 3 and 4 Only.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	57.7	54.1	60.2	57.6	1.4	54.8	59.6
4	72.4	67.5	80.5	72.5	2.4	68.7	77.4
3 + 4	59.9	56.5	62.0	59.7	1.2	57.2	61.5

Table D-33 – Detailed Results for the Function-Return-Diff-Addtl (M20) Technique Considering Versions 3 and 4 Only.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	60.8	57.9	63.5	60.9	1.2	58.1	62.9
4	63.1	57.0	71.3	63.3	3.4	57.3	69.6
3 + 4	61.2	58.6	63.3	61.2	1.1	59.0	63.2

Table D-34 – Detailed Results for the Block-Diff-Addtl (M21) Technique Considering Versions 3 and 4 Only.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	74.8	73.9	77.3	75.0	0.7	74.1	76.7
4	60.3	55.2	68.1	60.4	2.8	56.2	65.6
3 + 4	72.7	71.3	75.0	72.9	0.7	71.7	74.6

Table D-35 – Detailed Results for the Basic-Block-Diff-Addtl (M22) Technique Considering Versions 3 and 4 Only.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	78.3	77.2	80.7	78.4	0.7	77.4	80.4
4	62.1	56.4	68.9	62.2	2.9	57.1	68.0
3 + 4	76.0	74.8	78.0	76.1	0.7	75.1	77.6

Table D-36 – Detailed Results for the Decision-Diff-Addtl (M23) Technique Considering Versions 3 and 4 Only.

Software Version	APFD Score						
	MEDIAN	MIN	MAX	MEAN	STDEV	95% Confidence Interval	
						Lower Bound (2.5^{th} percentile)	Upper Bound (97.5^{th} percentile)
3	76.3	75.0	79.0	76.4	0.8	75.3	78.5
4	67.0	59.7	74.5	67.2	3.3	61.4	73.8
3 + 4	75.0	73.2	77.5	75.1	0.8	73.8	77.2

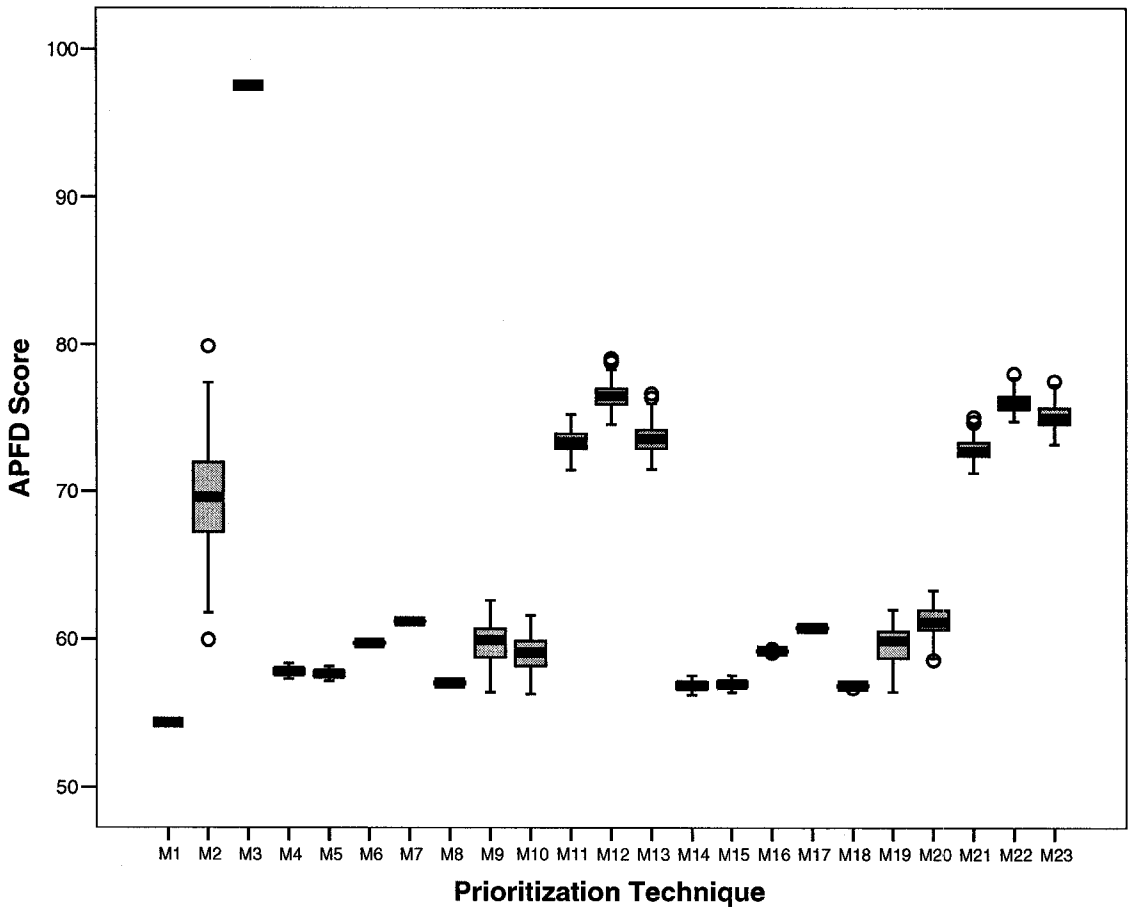


Figure D-5 – Weighted average APFD scores for each of the prioritization techniques across versions 3 and 4.