AN ABSTRACT OF THE THESIS OF

Amit Goel for the degree of Master of Science in Computer Science presented on December 2, 2002.

Title: An Experimental Study Of Cost Cognizant Test Case Prioritization

Redacted for Privacy

Abstract approved: _____

Gregg Rothermel

Test case prioritization techniques schedule test cases for regression testing in an order that increases their ability to meet some performance goal. One performance goal, rate of fault detection, measures how quickly faults are detected within the testing process. The APFD metric had been proposed for measuring the rate of fault detection. This metric applies, however, only in cases in which test costs and fault costs are uniform. In practice, fault costs and test costs are not uniform. For example, some faults which lead to system failures might be more costly than faults which lead to minor errors. Similarly, a test case that runs for several hours is much more costly than a test case that runs for a few seconds. Previous work has thus provided a second, metric $APFD_C$, for measuring rate of fault detection, that incorporates test costs and fault costs. However, studies of this metric thus far have been limited to abstract distribution models of costs. These distribution models did not represent actual fault costs and test costs for software systems.

In this thesis, we describe some practical ways to estimate real fault costs and test costs for software systems, based on operational profiles and test exe-

cution timings. Further we define some new cost-cognizant prioritization techniques which focus on the $APFD_C$ metric. We report results of an empirical study investigating the rate of "units-of-fault-cost-detected-per-unit-test-cost" across various cost-cognizant prioritization techniques and tradeoffs between techniques.

The results of our empirical study indicate that cost-cognizant test case prioritization techniques can substantially improve the rate of fault detection of test suites. The results also provide insights into the tradeoffs among various prioritization techniques. For example: (1) techniques incorporating feedback information (information from previous tests) outperformed those without any feedback information; (2) technique effectiveness differed most when faults are relatively difficult to detect; (3) in most cases, technique performance was similar at function and statement level; (4) surprisingly, techniques considering change location did not perform as well as expected. The study also reveals several practical issues that might arise in applying test case prioritization, as well as opportunities for future work.

An Experimental Study Of Cost Cognizant Test Case Prioritization

by

Amit Goel

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed December 2, 2002
Commencement June 2003

Master of Science thesis of Amit Goel presented on December 2, 2002

APPROVED:

Redacted for Privacy

Major Professor, representing Computer Science

Redacted for Privacy

Chair of the Department of Computer Science

Redacted for Privacy

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Amit Goel, Author

ACKNOWLEDGMENTS

I would especially like here to express my sincere thanks and strong feeling of gratitude to my major professor, Dr. Gregg Rothermel, for his enduring guidance, support and encouragement for completing this work. I learned a great deal about software regression testing from his encylopaedic knowledge. Thanks so much for his continuous reading and editing this report with such nice patience and big efforts.

Thanks also to Dr. Bella Bose, for serving as my minor professor. Dr. Bose is one of the kindest persons I know. He showed genuine interest in how the things are going with me. I would also like to thank Dr. Saurabh Sethia, for being on my committee.

Thanks to all the members of empirical software testing group, particulary to Alexey Malishevsky, to whom I always asked questions regarding various tools needed for our research. I would also like to thank Dan Chirica, Hyunsook Do, Dalai Jin, Joe Ruthruff, Desiree Dunn, and Adam Ashenfelter for their contributions in subject development needed for our research. Thanks also to all the other people who gave me help in this thesis.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDIX TABLES

AN EXPERIMENTAL STUDY OF COST COGNIZANT TEST CASE PRIORITIZATION

CHAPTER 1

INTRODUCTION

Software testing plays an important role in the software life-cycle. It is well known that there are still undetected errors in most popular software packages when they are released for real-life applications. Effective testing helps in revealing potential faults and ensuring system qualities.

No matter how well conceived and tested before being released, software will eventually have to be modified in order to correct faults or respond to changes in user requirements. *Regression testing* is testing performed to provide confidence that the quality of the system is maintained after such changes.

Software engineers often save the test suites they develop for their software so that they can reuse those test suites later for regression testing. Rerunning all of the test cases in a test suite, however, can be expensive. For example, it is reported that for one commercial avionics software system that contains about 20,000 lines of code, it would take seven weeks to run the entire test suite[4]. In cases such as this, testers may want to arrange their test cases such that those with highest priority, according to some criterion, are run earlier than those with low priority.

Researchers have looked at various techniques for reducing the cost of regression testing, including test selection and test suite minimization techniques. In general, however, these techniques can have drawbacks. For example, some empirical studies argue that there is little or no loss in the fault detecting capability of a minimized suite as compared to an unminimized original[18, 19], but other studies show that fault revealing capabilities can be severely compromised[15] by minimization. Similarly, although there are safe regression test selection techniques[1, 3, 14] that ensure adequacy of a selected test suite subset, the conditions under which safety can be achieved do not always hold[13, 14].

Test case prioritization techniques[16] provide another way to improve the cost effectiveness of regression testing. These techniques attempt to schedule test cases in an order that increases their effectiveness at meeting some performance goal. For example, test cases might be scheduled in an order that achieves code coverage at the fastest rate possible, exercises features in order of frequency of use, or reflects their historically observed abilities to detect faults.

One major goal of test case prioritization is that of increasing a test suite's *rate of fault detection*- that is, how quickly a test suite detects faults during the testing process. An improved rate of fault detection during the testing process provides earlier feedback on the system under test, allowing debugging to begin earlier, and supports faster strategic decisions about the release of the version. Moreover, an increased rate of fault detection can increase the likelihood that, if testing is prematurely halted, test cases that offer the greatest fault detection ability in the available testing time will have been executed.

The APFD metric[16] has been proposed to measure the average cumulative percentage of faults detected over the course of executing the test cases in a test suite in a particular order. It has been shown that the APFD metric can be

used to compare the rate of fault detection achieved by various prioritized test suites. Although successful in application to the class of problems for which it was designed, the APFD metric relies on the assumption that test costs and fault costs are uniform. In practice, however, test costs and fault costs can vary. For example, some faults which lead to system failures might be more costly than faults which lead to minor errors. Similarly, a test case that runs for several hours is much more costly than a test case that runs for a few seconds. When differences such as these occur, the APFD metric can assign inappropriate values to test case orders, and techniques designed to improve test case orders under that metric can produce unsatisfactory results.

Elbaum et al.[4] address this shortcoming of the APFD metric and suggest a new "cost-cognizant" APFD metric, $APFD_C$, that takes into consideration varying fault costs and test costs. Prioritization techniques that account for these varying costs are applied and measured against the new metric. The authors discuss several practical issues that arise in the use of varying test costs and fault costs, and illustrate differences between the previous and new metric.

Building on that work, this thesis addresses several additional questions. First, [4] examined the assignment of fault costs and test costs based on different distribution models. Those models did not represent actual fault costs or test costs occurring in practice. In this paper, we define ways to estimate real fault costs and test costs, based on operational profiles and test execution timings.

Second, only four prioritization techniques were examined in [4]. In this work we consider cost-cognizant adaptions of those four techniques, plus several others.

Finally, the empirical studies in [4] considered only a single relatively small program of 6 K lines of C code, with versions consisting only of small seeded

faults. In this work we focus on five larger programs, each with several real releases.

In this thesis, we describe our empirical study in which we investigate the ability of cost-cognizant prioritization techniques to improve the rate of fault detection as measured by the $APFD_C$ metric. The higher the rate of fault detection, the earlier feedback on costly faults is returned to the developer, and thus, the earlier the developer can begin debugging the code and fixing the faults. The results of our empirical study indicate that cost-cognizant test case prioritization techniques can substantially improve the rate of fault detection of test suites. Meanwhile, our results illustrate various effects of these prioritization techniques and highlight several tradeoffs between them.

In the next chapter, we present background information describing the test case prioritization problem, prioritization techniques, and the $APFD_C$ metric. Chapter 3 describes methods for determining fault costs and test costs; it also describes our cost-cognizant prioritization techniques. Chapter 4 presents the design and results of our empirical study. Chapter 5 discusses implications of those results. Chapter 6 draws conclusions and discusses future work.

CHAPTER 2

BACKGROUND

2.1 The Test Case Prioritization Problem

Rothermel et al.[16] defined the test case prioritization problem and described several issues relevant to its solution; this section reviews that work.

> *The Test Case Prioritization Problem:*
>
> *Given:* T, a test suite; PT, the set of permutations of T; f, a function from PT to the real numbers.
>
> *Problem:* Find $T' \in PT$ such that $(\forall T'')\ (T'' \in PT)\ (T'' \neq T')$ $[f(T') \geq f(T'')]$.

In this definition, PT is the set of all possible prioritizations (orders) of T, and f is the objective function that, applied to any such order, yields an *award value* for that order.

There are various goals that can be addressed in test case prioritization. Possible goals include the following:

- Testers may wish to increase the rate of fault detection of their test suites - that is, the likelihood of revealing faults earlier in the testing process than would be possible with an ad-hoc or random testing order.

- Testers may wish to increase the rate of detection of high-risk faults - that is, locate such faults earlier in the testing process than would be possible with an ad-hoc ordering.

- Testers may wish to increase the coverage of coverable code in the program under test at a faster rate than would be possible with an ad-hoc test ordering.

- Testers may wish to increase their confidence in the reliability of the system under test at a faster rate.

- Testers may wish to increase the likelihood of revealing faults related to specific code changes earlier in the regression testing process.

For each goal, an objective function needs to be defined to measure a prioritization technique's ability to meet the goal.

Rothermel et al. [16] also distinguish two types of test case prioritization: general and version-specific. In *general test case prioritization*, given program P and test suite T, test cases in T are prioritized with the goal of finding a test case order that will be useful over a sequence of subsequent modified versions of P. Thus, general test case prioritization can be performed following the release of some version of the program during off-peak hours, and the cost of performing the prioritization is amortized over the subsequent releases. The expectation is that the resulting prioritized suite will be more successful than the original suite at meeting the goal of the prioritization, *on average* over those subsequent releases.

In contrast, in *version-specific test case prioritization*, given program P and test suite T, test cases in T are prioritized with the intent of finding an ordering

that will be useful on a specific version P' of P. Version-specific prioritization is performed after a set of changes have been made to P and prior to regression testing P'. Because this prioritization is performed after P' is available, care must be taken to prevent the cost of prioritizing from excessively delaying the very regression testing activities it is supposed to facilitate. The prioritized test suite may be more effective at meeting the goals of the prioritization for P' in particular than would a test suite resulting from general test case prioritization, but may be less effective on average over a succession of subsequent releases.

Thus, given any prioritization goal, various *test case prioritization techniques* may be used to meet that goal. For example, to increase the rate of fault detection we might prioritize test cases in terms of the extent to which they execute modules that have tended to fail in the past. Alternatively, we might prioritize test cases in terms of greatest-to-least coverage-per-cost of features listed in a requirement specification. In any case, the intent behind the choice of a prioritization technique is to increase the likelihood that the prioritized test suite can better meet the goal than would an ad-hoc or random order of test cases.

In this work we are looking at *version-specific test case prioritization* in application to regression testing. We focus on a specific goal and function f, and we evaluate the abilities of several prioritization techniques to help us meet the goal.

2.2 Measuring Effectiveness

To measure how rapidly a prioritized test suite detects faults (the rate of fault detection of the test suite) we require an appropriate objective function f. For

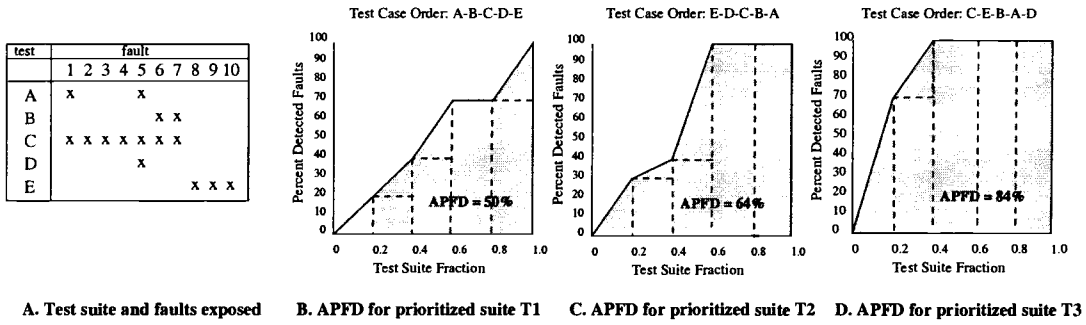


FIGURE 2.1: Examples illustrating the APFD metric.

this purpose, in [5, 16], the authors defined a metric, APFD. We discuss this metric briefly in the following section.

2.2.1 APFD metric

APFD is the weighted average of the percentage of faults detected during the execution of a test suite. APFD value ranges from 0 to 100; higher values imply faster (better) fault detection rates.

Consider an example program with 10 faults and a test suite of 5 test cases, **A** through **E**, with fault detecting abilities as shown in Figure 2.1.A. Suppose we place the test cases in order **A–B–C–D–E** to form prioritized test suite $T1$. Figure 2.1.B shows the percentage of detected faults versus the fraction of $T1$ used. After running test case **A**, 2 of the 10 faults are detected; thus 20% of the faults have been detected after 0.2 of $T1$ has been used. After running test case **B**, 2 more faults are detected and thus 40% of the faults have been detected after 0.4 of $T1$ has been used. In Figure 2.1.B, the area inside the inscribed rectangles (dashed boxes) represents the weighted percentage of faults detected over the corresponding fraction of the test suite. The solid lines connecting the corners

of the inscribed rectangles delimit the area representing the gain in percentage of detected faults. The area under the curve represents the weighted average of the percentage of faults detected over the life of the test suite. This area is the prioritized test suite's average percentage faults detected metric (APFD); the APFD is 50% in this example.

Figure 2.1.C reflects what happens when the order of test cases is changed to **E–D–C–B–A**, yielding a "faster detecting" suite than $T1$ with APFD 64%. Figure 2.1.D shows the effects of using a prioritized test suite $T3$ whose test case order is **C–E–B–A–D**. By inspection, it is clear that this order results in the earliest detection of the most faults and illustrates an optimal order, with APFD 84%.

2.2.2 Limitations of the APFD metric

Elbaum et al.[4] defined the limitations of the APFD metric and described a new APFD metric; this section and the following section review that work.

The APFD metric presented in the above section relies on two assumptions: (1) all faults have equal severity, and (2) all test cases have equal costs. (These assumptions are evident in the fact that the metric simply plots the percentage of faults detected against the fraction of the test suite run.) In [5, 16], the authors suggest that when these assumptions hold, the metric operates well. In practice, however, there are cases in which these assumptions do not hold: cases in which faults vary in severity and test cases vary in cost. In such cases, the APFD metric can provide unsatisfactory results, as the following simple examples illustrate.

Example 1. Consider the testing scenario illustrated in Figure 2.1. Under the APFD metric, when all ten faults are equally severe and all five test cases are equally costly, orders **A–B–C–D–E** and **B–A–C–D–E** are equivalent in terms of rate of fault detection; swapping **A** and **B** alters the rate at which *particular* faults are detected, but not the overall rates of fault detection. This equivalence is reflected in equivalent APFDs (50%). Suppose, however, that **B** is twice as costly as **A**; requiring two hours to execute where **A** requires one. In terms of faults-detected-per-hour, test case order **A–B–C–D–E** is preferable to order **B–A–C–D–E**, resulting in faster detection of faults. The APFD metric, however, does not distinguish between the two orders.

Example 2. Again working with the scenario given in Figure 2.1, suppose that all five test cases have equivalent costs, and suppose that faults 2-10 have severity k, while fault 1 has severity $2k$. In this case, test case **A** detects this more severe fault along with one less severe fault, whereas test case **B** detects only two less severe faults. In terms of fault-severity-detected, test case order **A–B–C–D–E** is preferable to order **B–A–C–D–E**. Again, the APFD metric does not distinguish between these two orders.

Example 3. Examples 1 and 2 provide cases in which the APFD metric proclaims two orders *equivalent* where our intuitions say they are not. It is also possible, when test costs or fault costs differ, for the APFD metric to assign a *higher* value to a test case order that we would consider *less* valuable. Working again with Figure 2.1, suppose that all ten faults are equally severe, and that test cases **A**, **B**, **D**, and **E** each require one hour to execute, but test case **C** requires ten hours. Consider test case order **C–E–B–A–D**. Under the APFD metric this order is assigned an APFD value of 84% (see Figure 2.1.D). Consider alternative test

case order **E–C–B–A–D**. This order is illustrated with respect to the APFD metric in Figure 2.2; because that metric does not differentiate test cases in terms of relative costs, all vertical bars in the graph (representing individual test cases) have the same width. The APFD for this order is 76%, lower than the score for test case order **C–E–B–A–D**. However, in terms of faults-detected-per-hour, the second order (**E–C–B–A–D**) is preferable: it detects 3 faults in the first hour, and remains better in terms of faults-detected-per-hour than the first order up through the end of execution of the second test case.

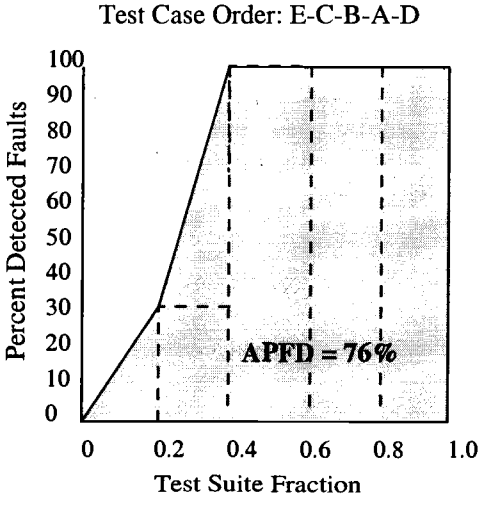


FIGURE 2.2: APFD for Example 3.

An analogous example can be created by using varying fault costs while holding test case costs uniform.

Example 4. Finally, consider an example in which both fault costs and test case costs vary. Suppose that test case **B** is twice as costly as test case **A**, requiring two hours to execute where **A** requires one. In this case, in Example

1, assuming that all ten faults were equally severe, we found test case order **A–B–C–D–E** preferable. However, if the faults detected by **B** are more costly than the faults detected by **A**, order **B–A–C–D–E** may be preferable. For example, suppose test case **A** has cost "1", and test case **B** has cost "2". If faults 1 and 5 (the faults detected by **A**) are assigned severity "1", and faults 6 and 7 (the faults detected by **B**) are assigned costs greater than "2", then order **B–A–C–D–E** achieves greater "units-of-fault-severity-detected-per-unit-test-cost" than does order **A–B–C–D–E**. [1] Again, the APFD metric would not make this distinction.

2.2.3 Cost-cognizant APFD metric

The examples in the previous section suggest that, when considering the relative merits of test cases, a measure for rate-of-fault-detection that assumes that test costs and fault costs are uniform can produce unsatisfactory results.

The notion that a tradeoff exists between the costs of testing and the costs of leaving undetected faults in software is fundamental in practice. Testers have to decide about this tradeoff frequently in practice. Therefore, it is appropriate to consider this tradeoff when prioritizing test suites. A metric for evaluating test case orders must accommodate test costs and fault costs. One approach to accomodating this is to require such a metric to reward test case orders proportionally to their "units-of-fault-cost-detected-per-unit-test-cost".

---

[1] In this example, for simplicity, a "unit" of fault severity is assumed to be equivalent to a "unit" of test case cost. Clearly, in practice, the relationship between fault severity and test case cost will vary among applications, and quantifying this relationship may be non-trivial. This is discussed further in the following section.

In [4], the authors have defined such a "cost-cognizant" metric by adapting the APFD metric; the new metric is called $APFD_C$. In terms of the graphs used in Figures 2.1 and 2.2, creation of this new metric entails two modifications. First, the horizontal axis is changed to denote "Percentage Total Test Case Cost Incurred" instead of "Test Suite Fraction". Now, each test in the test suite is represented by an interval along the horizontal axis, whose length is proportional to the percentage of total test suite cost accounted for by that test case. Second, the vertical axis is changed to denote "Percentage Total Fault Cost detected" instead of "Percent Detected Faults". Now, each fault detected by the test suite is represented by an interval along the vertical axis, whose height is proportional to the percentage of total fault cost that fault accounts for.

Following this new interpretation, in graphs such as the APFD graphs we have shown, a test's contribution is "weighted" along the horizontal axis in terms of test cost, and along the vertical axis in terms of the cumulative cost of the faults it reveals. In such graphs, the curve depicts a greater area for a test case order that has greater "units-of-fault-cost-detected-per-unit-test-cost".

To futher explain the $APFD_C$ metric from this graphical view point, Figure 2.3 presents graphs for each of the four examples presented in the preceding subsection. The leftmost pair of graphs (Figure 2.3.A) correspond to Example 1: the upper of the pair represents the $APFD_C$ for test case order **A–B–C–D–E**, and the lower represents the $APFD_C$ for order **B–A–C–D–E**. Note that whereas the (original) APFD metric did not distinguish the two orders, the $APFD_C$ metric gives preference to the order that places the less expensive test first, **A–B–C–D–E**.

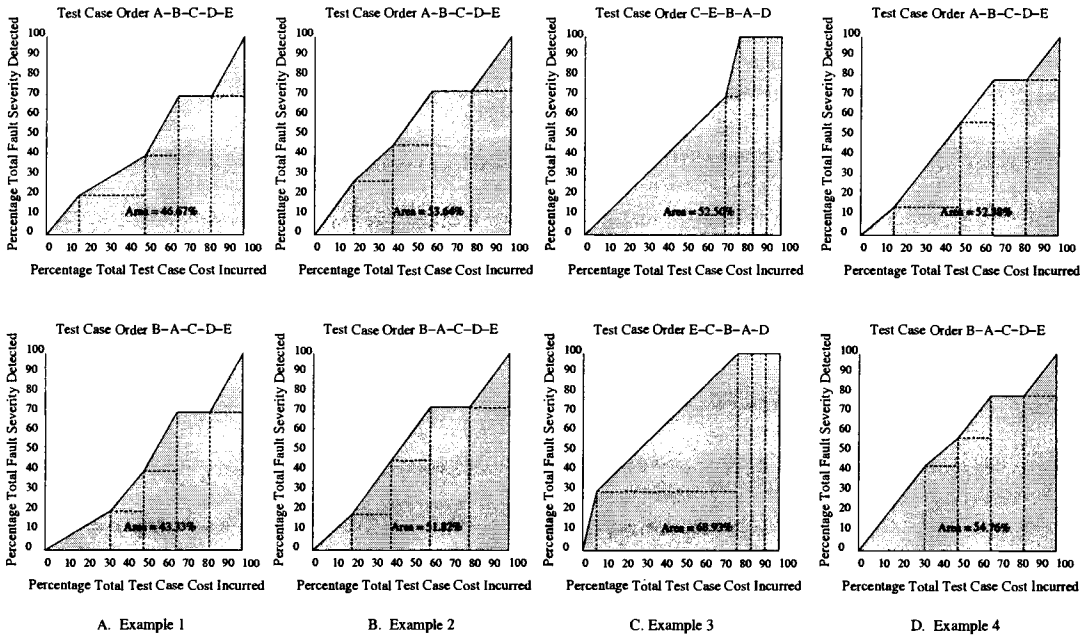


FIGURE 2.3: Examples illustrating the APFD_C metric.

Similarly, the other pairs of graphs illustrate the application of the APFD_C metric in Examples 2, 3, and 4. The pair of graphs in Figure 2.3.B, corresponding to Example 2, show how the new metric gives a higher award to the test case order that reveals the more severe fault earlier (**A–B–C–D–E**), under the assumption that the cost value assigned to faults 2-10 is 1 and the cost value assigned to fault 1 is 2. The pair of graphs in Figure 2.3.C, corresponding to Example 3, show how the new metric distinguishes test case orders involving a high-cost test case **C**: instead of undervaluing order **E–C–B–A–D**, the metric now assigns it greater value than order **C–E–B–A–D**. Finally, the pair of graphs in Figure 2.3.D, corresponding to Example 4, show how the new metric distinguishes between test case orders when both test case costs and fault severities are non-uniform, under the assumptions that test case B has cost 2 while all other test cases have cost 1, and that faults 6 and 7 have cost 3 while

all other faults have cost 1. In this case, the new metric assigns a greater value to order **B–A–C–D–E** than to order **A–B–C–D–E**.

The APFD_C metric can be quantitatively defined as follows. Let T be a test suite containing n tests 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 in an ordering T' of T that reveals fault i. The (cost-cognizant) weighted average percentage of faults detected during the execution of T' is given by the equation:

$$\text{APFD}_C = \frac{\sum_{i=1}^{m} (f_i \times (\sum_{j=TF_i}^{n} t_j - \frac{1}{2} t_{TF_i}))}{\sum_{i=1}^{n} t_i \times \sum_{i=1}^{m} f_i} \tag{2.1}$$

Equation 2.1 remains applicable when either test costs or fault costs are identical. Further, when both test case costs and fault severities are identical, the formula reduces to the formula for APFD.

A final issue to consider, with respect to APFD_C, involves calculating fault costs and test costs. We address this issue in the next chapter.

CHAPTER 3

DETERMINING TEST COSTS AND FAULT COSTS

3.1 Operational Profiles

To understand the concept of an *operational profile*, we first define the term "operation". An operation is an externally-initiated task performed by a system "as built"[10]. It can be compared with a function, which is an externally-initiated task to be performed by the system as viewed by users. The need for such a task first occurs in the mind of users, who forward it to software engineers as a requirement. At this stage it is a function. As the system is developed by software developers, functions evolve into and are implemented as operations. Functions often map one-to-one to operations, but the mapping is also often more complex, driven by performance and other needs. Examples of operations include specific commands, transactions, and processing of external events.

An *operational profile* is simply a set of operations and their probability of occurrence[9, 10]; in other words, given system S, the operational profile for S is the list of things users can do with S, and the frequency with which they do them. For example, working with a simple Text Editor, the "change font" operation may account for 20%, the "cut" operation for 10%, the "text insertion" operation for 50%, "copy" operations for 10%, and the "paste" operation for 10%. Table 3.1 shows the corresponding operational profile for such a Text Editor.

Operation	Occurence Probability
Change Font	.20
Cut	.10
Text Insertion	.50
Copy	.10
Paste	.10

TABLE 3.1: Sample Text Editor Operational Profile.

Operational profiles benefit a wide variety of activities associated with development of software-based systems, including system engineering, system design, development, testing, and operational use[9, 10]. For instance one use of operational profiles can be to test the operations in proportion to the frequency with which they occur. For example, to test the Text Editor under the profile presented above, we would spend half of our effort on testing operation "Text Insertion".

In this work we use operational profiles to assess *fault costs* and *test criticality*. For our subjects, first we identify the set of "operations" users perform with the subjects. Then we associate operations with the test cases in our test suites for the subjects. This lets us make an "Operation Matrix" for a subject, pairing operations with tests. For example, a sample "Operation Matrix" for the Text Editor example is given in Table 3.2. In the Operation Matrix, a '1' indicates that an operation is used by a test case and a '0' indicates that it is not. For example, in Table 3.2, Test1 uses all operations except "Copy"and "Paste".

Operation	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8
Change Font	1	1	1	0	0	0	0	0
Cut	1	1	1	0	0	0	0	0
Text Insertion	1	1	1	0	0	0	0	0
Copy	0	0	0	1	1	1	1	1
Paste	0	0	0	1	1	1	1	1

TABLE 3.2: Sample Operation Matrix for Text Editor.

The Operation Matrix is used to determine *fault costs* and *test criticality* by a procedure described in the following sections.

3.2 Test Criticality

We have defined test criticality as how important a test is for users of the software system. Test criticality can be associated with the frequency of usage of different operations identified in the system. A test using more frequent operations in the software system is more critical than a test using less frequent operations.

Test criticality is the sum of the probability of occurrence of various operations used by the test case. For example in Table 3.2, the test criticality of Test1 is the sum of the frequencies of usage of the operations "Change Font", "Cut", and"Text Insertion", i.e .20+.10+.50 = 0.80. (The values of probability of occurrence of different operations used in calculating the test criticality of Test1 are retrieved from Table 3.1). It is evident from Table 3.2 that Test1 is

more critical than Test4 to Test8 in the table, in the sense that Test1 is more likely to involve operations that matter most to the users of the software system.

3.3 Test Costs

Test cost is the time required to execute a test. When prioritizing for regression testing, prioritization techniques can take advantage of timings gathered in the previous execution of the test suite. Of course, this timing data may not reflect precisely the test costs that will occur in the future after the program under test has been modified, so we must be satisfied with estimates. It seems reasonable, however, to expect that historical test cost data can be used to predict future test costs with sufficient accuracy. In practice such estimates are often available[11].

3.4 Fault Costs

Predicting fault costs in practice is much more difficult. Several techniques, however, have been proposed for use in doing so[10, 11, 17]. Musa[10] introduces guidelines for the definition of "failure severity classes" by which each system operation can be associated with a class of failures, usually based on experience and historical data. Musa's goal was to determine failure intensity for the product under consideration. More consistent classifications arise from software with high safety requirements, where each failure event must have a probability and a risk level (e.g, catastrophic, critical, or negligible). The criticality of each component is then derived from its association with a set of failure events[17].

In this work, however, we use operational profiles to estimate fault cost. Our thesis is that the failures present in the operations with high frequency of use are most likely to be encountered when the software is in use, and hence

Fault	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8
Fault 1	1	1	1	0	0	0	0	0
Fault 2	1	0	0	0	0	0	0	0
Fault 3	0	0	0	1	1	1	1	1

TABLE 3.3: Sample Fault Matrix for Text Editor.

would be more problematic to the users. Thus, these types of failure should be assigned greater cost than failures which are present in the relatively low use operations.

To determine fault costs for our subjects, we use the linkage between the Fault Matrix and Operation Matrix. The Fault Matrix gives the pairing between tests and faults, i.e, it lists all tests exposing each fault. A Fault Matrix for the Text Editor example is given in Table 3.3. As evident from the Table, there are three faults in the Text Editor. In the Fault Matrix, a '1' indicates that a fault is revealed by a test case and a '0' indicates that it is not.

We have used two heuristics to determine the fault costs. These algorithms or heuristics are described in the following sections.

3.4.1 Fault cost algorithm 1

In this algorithm, we first determine the association between faults and operations. This link is determined by the Fault Matrix and the Operation Matrix. From Table 3.3 we can determine which fault is revealed by which test, and then, looking at the Operation Matrix (Table 3.2), we can determine which

Fault	Change Font	Cut	Text Insertion	Copy	Paste
Fault 1	1	1	1	0	0
Fault 2	1	1	1	0	0
Fault 3	0	0	0	1	1

TABLE 3.4: Sample Fault-Operation Matrix for Text Editor.

operation is used by which test. Hence we can determine which operations are affected by each fault. Table 3.4 gives the "Fault-Operation Matrix" for the Text Editor example. '1' indicates that the operation is affected by the fault and a '0' indicates that it is not.

Now, the fault cost is simply the sum of the probability of occurrence of operations affected by the fault; for example, the *cost of fault X is the sum of probabilities of operations affected by X*, and equals Change Font_p + Cut_p + Text Insertion_p = .20 + .10 + .50 = 0.80 (from Table 3.1). Similarly, the cost of fault 2 is 0.80, and the cost of fault 3 is 0.20.

3.4.2 Fault cost algorithm 2

Algorithm 1 assigns the same costs to Fault 1 and Fault 2. Fault 1 is revealed by Test1, Test2, and Test3 while Fault 2 is revealed only by Test1. Though all three tests perform the same operations, these tests use different inputs to perform the operations. If tests are designed in relation to usage patterns, if the user is performing the "Text Insertion" operation on Text Editor, then the probability of encountering Fault 1 is greater than the probability of encountering Fault 2,

since Fault 1 is revealed by three sets of inputs while Fault 2 is revealed by only a single set of inputs. Hence the cost of Fault 1 should be higher than the cost of Fault 2.

To accomodate this case, we defined one more heuristic which takes into account the number of tests associated with each operation. If an operation is affected by two faults, f1 and f2, and if f1 is exposed by more tests than f2, we assign greater cost to f1 than f2.

One way to assign greater cost to Fault 1 than Fault 2 is to simply take the number of tests as a weight in calculation of fault cost. Let T_i be a list of test cases associated with operation i. Let Op be the set of m operations affected by fault X, and let P_i be the probability associated with operation i. The cost of fault X can be defined as:

$$FaultCost_X = \sum_{1 \le i \le m} P_i \times T_i \tag{3.1}$$

By this technique the costs of Fault 1, Fault 2, and Fault 3 in our example are :

Fault 1 cost $= 3 \times 0.20 + 3 \times 0.10 + 3 \times 0.50 = 2.40$

Fault 2 cost $= 0.20 + 0.10 + 0.50 = .80$

Fault 3 cost $= 5 \times 0.10 + 5 \times 0.10 = 1.0$

Although this technique assigns higher cost to Fault 1, it also makes Fault 3 more costly than Fault 2; this is inappropriate because Fault 2 can be present in the operation which has the highest probability of occurrence, "Text Insertion", while Fault 3 will never be encountered while using operation "Text Insertion". This contradicts our thesis that faults occurring in the most frequently used operations would be more costly than faults occurring in less frequently used operations. Therefore, there should be some threshold value above which the number of tests would be considered and below which it would not. This thresh-

old value is determined by the average of the probability of occurrence of the highest probability operation affected by the fault and the lowest probability operation affected by the fault. Let P_H be the probability of occurrence of the highest probability operation affected by the fault and P_L be the probability of occurrence of the lowest probability function affected by the fault. The threshold value for TH is given by the equation:

$$TH = \frac{P_H + P_L}{2} \tag{3.2}$$

Hence the threshold value for the Text Editor example is:

$TH = \frac{0.50+0.10}{2} = 0.30$

Therefore the fault costs based on this threshold value are:

Fault 1 cost $= 0.20 + 0.10 + 3 \times 0.50 = 1.80$

Fault 2 cost $= 0.20 + 0.10 + 0.50 = 0.80$

Fault 3 cost $= 0.10 + 0.10 = 0.20$

3.5 Cost-Cognizant Prioritization Techniques

Our main focus in this work is to increase the likelihood of revealing more costly faults earlier in the testing process, while also accounting for test costs. Informally, we describe this goal as one of improving our test suite's *rate of fault cost detection.*

In this work, we consider nine different *cost-cognizant* prioritization techniques. Several of these techniques were considered in [5, 16] but here they are adapted to consider varying test costs and fault costs. We next describe the nine techniques in turn.

T1: Random Ordering (random).

As an experimental control, one prioritization "technique" that we consider is to randomly order the tests in the test suite. Such an ordering makes no attempt to account for varying test costs and fault costs.

T2: Total Statement Coverage Prioritization (tcov-stat).

Using program instrumentation we can measure the coverage of statements in a program by its tests. We can then prioritize tests in terms of the total number of statements they cover, by counting the number of statements covered by each test, and then sorting the tests in descending order of that number.

To adapt this technique to the case in which test costs and fault costs vary, instead of sorting by the total number of statements covered by test j, we find the value of $\frac{test-criticality_j \times statements_j}{test-cost_j}$ for each test j, where $test\text{-}criticality_j$ is the criticality of test j, $test\text{-}cost_j$ is the cost of test j, and $statements_j$ is the total number of statements covered by test j. Then we sort the tests in descending order of the above value. The notion behind this ratio of criticality to cost is to reward test cases that have greater ratios of fault-cost-detected per unit-test-cost.

T3: Total Function Coverage Prioritization (tcov-func).

Analogous to total statement coverage prioritization but operating at the level of functions, this technique prioritizes tests according to the total number of functions covered by them. To adapt this technique to the case in which test costs and fault costs vary, we find the value $\frac{test-criticality_j \times functions_j}{test-cost_j}$ for each test j, where $test\text{-}criticality_j$ is the criticality of test j, $test\text{-}cost_j$ is the cost of test j, and $functions_j$ is the total number of functions covered by test j. Then we sort the tests in descending order of the above value.

T4: Additional Statement Coverage Prioritization (acov-stat).
Additional statement coverage prioritization is like total coverage prioritization, but it relies on feedback about coverage attained so far in testing to focus on statements not yet covered. To do this, the technique greedily selects a test that yields the greatest statement coverage, then adjusts the coverage data about subsequent test cases to indicate their coverage of statements not yet covered, and then repeats this process until all statements covered by at least one test have been covered. If test cases remain, the process is repeated.

To adapt this technique to the case in which test costs and fault costs vary, instead of selecting tests with the highest number of additional statements covered, we calculate the value $\frac{test-criticality_j \times additional-statements_j}{test-cost_j}$ for each test j, where $test\text{-}criticality_j$ is the criticality of test j, $test\text{-}cost_j$ is the cost of test j, and $additional\text{-}statements_j$ is the total number of additional statements covered by test j. Then we prioritize in terms of the highest such values calculated.

T5: Additional Function Coverage Prioritization (acov-func).
Analogous to additional statement coverage prioritization but operating at the level of functions, this technique prioritizes tests according to the additional number of functions covered. To adapt this technique to the case in which test costs and fault costs vary, we do the same as with additional statement coverage but this time instead of using $additional\text{-}statements_j$, we use $additional\text{-}functions_j$ covered by test j, i.e, we find the value $\frac{test-criticality_j \times additional-functions_j}{test-cost_j}$ for each test and prioritize in terms of the highest values calculated.

T6: Total Binary Diff Function Coverage Prioritization(fi-tcov-func-bdiff).

Faults are not equally likely to exist in each function; rather, certain functions are more likely to contain faults than others. This fault proneness can be associated with measurable software attributes[2, 7, 8]. In the context of regression testing, we are also interested in the potential influence, on fault proneness, of our modifications; that is with the potential of modifications to lead to regression faults. The UNIX diff command can be used to estimate fault proneness.

With binary diff based techniques, for each function present in both P and P′, we measure the change in the output of the UNIX diff command applied to P and P′. We assign the binary value '1' to the function if there is a syntactic difference, otherwise we assign '0'. Given this measure, total binary diff function coverage prioritization is performed in a manner similar to total function coverage prioritization. For each test, we compute the sum of the binary values obtained by the UNIX diff command for every function that test executes. Then, we sort tests in decreasing order of these sums.

To adapt this technique to the case in which test costs and fault costs vary, instead of selecting tests with the highest sum of binary diff values, we calculate the value $\frac{test-criticality_j \times sum-binary-values_j}{test-cost_j}$ for each test j, where $test\text{-}criticality_j$ is the criticality of test j, $test\text{-}cost_j$ is the cost of test j, and $sum\text{-}binary\text{-}values_j$ is the sum of the binary diff values associated with each function that test executes. Then we prioritize the tests in terms of the highest such values calculated.

T7: Total Binary Diff Statement Coverage Prioritization(fi-tcov-stat-bdiff).

Analogous to total binary diff function coverage prioritization but operating at the level of statements, this technique prioritizes tests according to the sum of the binary diff values for every statement executed by the test. The binary diff data for the statement is approximated with the help of binary diff values for the function. The binary diff value of statement s in function f is calculated as $\frac{bd_f}{n_f}$, where bd_f is the binary diff value of function f, and n_f is the total number of statements in function f.

To adapt this technique to the case in which test costs and fault costs vary, instead of selecting tests with the highest sum of binary diff values of statements executed, we calculate the value $\frac{test-criticality_j \times sum-binary-values_j}{test-cost_j}$ for each test j, where $test\text{-}criticality_j$ is the criticality of test j, $test\text{-}cost_j$ is the cost of test j, and $sum\text{-}binary\text{-}values_j$ is the sum of the binary diff values for every statement that test j executes. Then we prioritize the tests in terms of the values calculated.

T8: Additional Binary Diff Function Coverage Prioritization(fi-acov-func-bdiff).

Additional binary diff function coverage prioritization is accomplished in a manner similar to additional function coverage prioritization, by incorporating feedback into total binary diff function coverage prioritization. The set of functions that have been covered by previously executed tests is maintained. To find the next best test we compute, for each test, the sum of the binary values for each additional function that test executes. The test for which this sum is the greatest wins. This process is repeated until all tests are prioritized.

To adapt this technique to handle varying test costs and fault costs, instead of selecting tests with the highest sum of binary values, we calculate the value $\frac{test-criticality_j \times sum-binary-value_j}{test-cost_j}$ for each test j, where $test\text{-}criticality_j$ is the criticality of test j, $test\text{-}cost_j$ is the cost of test j, and $sum\text{-}binary\text{-}value_j$ is the sum of binary values for every function covered additionaly by the test j. Then we prioritize the tests in terms of the highest such values calculated.

T9: Additional Binary Diff Statement Coverage Prioritization(fi-acov-stat-bdiff).

Analogous to additional binary diff function coverage prioritization but operating at the level of statements, this technique prioritizes tests based on the additional statements covered. The binary diff data for each statement is calculated in the same way as in technique T6. To adapt this technique to handle varying test costs and fault costs, instead of selecting tests with the highest sum of binary diff data, we calculate the value $\frac{test-criticality_j \times sum-binary-values_j}{test-cost_j}$ for each test j, where $test\text{-}criticality_j$ is the criticality of test j, $test\text{-}cost_j$ is the cost of test j, and $sum\text{-}binary\text{-}values_j$ is the sum of the binary diff values for every statement covered additionally by test j. Then we prioritize in terms of the values calculated.

CHAPTER 4

EMPIRICAL STUDY AND RESULTS

To further investigate cost-cognizant prioritization, and to compare and evaluate the cost-cognizant prioritization techniques described in Chapter 3, we performed a controlled experiment. This chapter describes the experiment, including design, materials, results, and threats to validity.

4.1 Research Questions

In our empirical studies, we are investigating the following research questions:

Q1: Can cost-cognizant prioritization improve the rate of "units-of-fault-cost-detected-per-unit-test-cost"?

Q2: How do cost-cognizant prioritization techniques perform in terms of rate of "units-of-fault-cost-detected-per-unit-test-cost" as *fault-hardness* varies?

Q3: How do the cost-cognizant prioritization techniques incorporating feedback information compare to the ones without feedback information in terms of rate of "units-of-fault-cost-detected-per-unit-test-cost"?

Q4: How do fine granularity (statement level) cost-cognizant prioritization techniques compare to coarse granularity (function level) techniques in terms of rate of "units-of-fault-cost-detected-per-unit-test-cost"?

Q5: Can the use of predictors of fault proneness improve the rate of "units-of-fault-cost-detected-per-unit-test-cost" of cost-cognizant prioritization techniques?

Q6: How does fault cost algorithm 1 compare to fault cost algorithm 2 in terms of its support for prioritization?

4.2 Experiment Measures

4.2.1 Independent variables

Our experiment manipulated the following three independent variables :

- The cost-cognizant prioritization technique (nine prioritization techniques, each explained in Chapter 3).
- Fault hardness. We consider three levels of fault hardness: *hard*, *moderate*, and *easy*. Each of these is defined in Section 4.3.3.
- Fault cost algorithm (two fault cost algorithms, as described in Chapter 3).

4.2.2 Dependent Variables

To investigate our research questions we need to measure the rate of "units-of-fault-cost-detected-per-unit-test-cost". Hence the dependent variable for our experiment is the metric, $APFD_C$, discussed in Chapter 2.

Program	Version	Lines of C Code	Regression Faults
grep	0	10,929	0
grep	1	12,654	4
grep	2	13,230	3
grep	3	13,373	3
grep	4	13,359	1
flex	0	11,783	0
flex	1	12,323	5
flex	2	14,034	4
flex	3	14,074	8
flex	4	14,171	1
make	0	18,665	0
make	1	19,902	5
make	2	20,678	5
make	3	21,872	3
make	4	25,465	4
sed	1	8,063	0
sed	2	11,911	5
sed	3	9,978	4
xearth	1	24,179	0
xearth	2	13,165	4
xearth	3	14,068	3

TABLE 4.1: Experiment Subjects.

4.3 Experiment Materials

4.3.1 Programs

The subject programs used in our experiment were five C programs (see Table 4.1). Three of these programs, *make*, *grep*, and *sed*, are open source applications developed as a part of the GNU Project. The fourth program, *xearth*, was developed by Kirk Lauritz Johnson of M.I.T. The fifth program, *flex*, was developed by Lawrence Berkeley Laboratory at University of California.

We explain the characterstics of these programs briefly in the following paragraphs. [1]

Grep.

Grep searches one or more input files for lines containing a match to a specified pattern. For this experiment we used five versions of *grep* (see Table 4.1). Each release corrects faults, but also provides new functionality as evident by increasing code size.

Flex.

Flex is a tool for generating scanners: programs which recognize lexical patterns in text. *Flex* reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. *Flex* generates as output a C source file, lex.yy.c, which defines a routine yylex(). This file is compiled and linked with the -lfl library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code. For this experiment we used five versions of *flex* (see Table 4.1).

Make.

Make controls the generation of executables and other non-source files of a program from the program's source files. *Make* gets its knowledge of how to build a program from a file called the "makefile", which lists each of the non-

---

[1] Two of these programs, flex and grep, were also used in earlier studies reported in [4]; here, we re-use these materials to investigate different research questions.

source files and how to compute it from other files. For this experiment we used five versions of *make* (see Table 4.1).

Sed.

Sed is a stream editor. A stream editor is used to perform basic text transformations on an input stream (a file or input from a pipeline). While in some ways similar to an editor which permits scripted edits (such as ed), *sed* works by making only one pass over the input(s), and is consequently more efficient. For this experiment we used three versions of *sed* (see Table 4.1).

Xearth.

Xearth sets the X root window to be an image of the Earth, as seen from any vantage point in space, correctly shaded for the current position of the Sun. By default, *xearth* updates the displayed image every five minutes; the time between updates can be changed using either X resources or a command line option. *Xearth* can also render its output directly into PPM and GIF files instead of drawing in the root window. For this experiment we used three versions of *xearth* (see Table 4.1).

4.3.2 Test suites

To examine our research questions we required a *test suite* for each of our subject programs. To create *test suites* for our programs, first we created specification-based tests using the category-partition method[12], and then we developed additional coverage based tests. Table 4.2 shows the test suite size for each program.

Program	Test Suite Size
grep	810
flex	568
make	1044
sed	1294
xearth	540

TABLE 4.2: Tests per Subject.

4.3.3 Faults

We wished to evaluate the performance of cost-cognizant prioritization techniques with respect to detection of regression faults—faults created in a program version as a result of the modifications that produced that version. To obtain such faults for our subjects, we asked several graduate and undergraduate computer science students, each with at least two years experience programming in C and unacquainted with the details of the study, to become familiar with the programs and to insert regression faults into the program versions. These 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.

Given ten potential faults seeded in each version of each program, we activated these faults individually, and executed the test suite for the programs to determine which faults could be revealed by which test. We excluded any potential faults that were not detected by any test: such faults are meaningless to $APFD_C$ measures and cannot influence our results. We also excluded any faults that were detected by more than 20% of the tests; our assumption was that such easily detected faults would be detected by test engineers during their

unit testing of modifications. The number of faults remaining after exclusions, and utilized in the studies, are reported in Table 4.1.

To further investigate the performance of cost-cognizant prioritization techniques with respect to the hardness of the regression faults, we categorized faults into three classes: *hard*, *moderate*, and *easy*. We categorized a fault as *hard* when the fault is detected by less than or equal to 1.25% of the tests for that subject program. A fault lies in the *moderate* class when it is detected by less than or equal to 5% of the tests. The *easy* class contains faults which are detected by less than or equal to 20% of the tests. The number of faults in each fault class across each version of each program is shown in Table 4.3.

4.3.4 Tools and supporting data

To perform the experiments we required several tools. Our test coverage and control-flow graph information was provided by the Aristotle program analysis system [6]. We created prioritization tools implementing the techniques outlined in Chapter 3. To determine test costs, test criticality, and fault costs, several tools were created. The procedures to calculate test criticality and fault costs were those defined in Chapter 3. To calculate test costs for our subjects, we included timestamps in test script outputs to calculate test timings in seconds.

To identify the various operations for our subjects, we employed several graduate and undergraduate computer science students, each of whom had been involved in the subject development and had thorough knowledge of a subject program. Then, since we could not obtain real operational profiles for our subjects, we randomly assigned probabilities of usage to the different operations identified.

Program	Version	Hard	Moderate	Easy
grep	0	-	-	-
grep	1	4	4	4
grep	2	1	3	3
grep	3	2	3	3
grep	4	1	1	1
flex	0	-	-	-
flex	1	-	1	5
flex	2	2	4	4
flex	3	4	4	8
flex	4	-	-	1
make	0	-	-	-
make	1	-	-	5
make	2	-	-	5
make	3	-	-	3
make	4	-	-	4
sed	1	-	-	-
sed	2	3	4	5
sed	3	1	3	4
xearth	1	-	-	-
xearth	2	1	2	4
xearth	3	1	2	3

TABLE 4.3: Faults per Fault Class.

4.4 Experiment Design

The experiment used a $9 \times 3 \times 2$ factorial design. That is, for each subject program P we applied the nine cost-cognizant prioritization techniques across all three fault hardnesses (*hard*, *moderate*, *easy*), calculating fault costs based on our two heuristics. Then, the $APFD_C$ values of the prioritized suites were evaluated and used as the statistical data.

4.5 Threats to Validity

In this section we describe the internal, external, construct, and conclusion threats to the validity of our experiments, and approaches we used to limit their impact.

Internal Validity.

To test our hypotheses we had to conduct experiments requiring a large number of processes and tools. Some of these processes involved programmers (e.g., creating operational profiles, fault seeding) and some of the tools were specifically developed for the experiments, all of which could have added variability to our results increasing threats to internal validity.

We used several procedures to control and minimize these sources of variation. For example, the fault seeding process was performed following a specification so that each programmer operated in a similar way, and it was performed in two locations using different programmers. Similarly, in the case of developing operational profiles, programmers followed the same specification. Also, we validated new tools by testing them on small programs and test suites, refining them as we targeted the larger subjects, and cross validating them across labs. Also, to minimize threats to the validity of timing data for the subject programs, we ran all the subject programs on identical machines with restricted access.

External Validity.

The threats to external validity of our studies are centered around the issue of how representative the subjects, faults, and test suites used in our studies are.

The subject programs considered in our study are medium size programs. Complex industrial programs with different characteristics may be subject to different cost-benefit tradeoffs. Our fault seeding process helped us control for threats to internal validity; however, faults and fault patterns may differ in practice. Other threats involve our testing process and test suites. If the testing process we used is not representative of industrial ones, the results might not generalize. Furthermore, test suite constitution is also likely to differ under different processes. Control of these threats can be achieved only through additional studies using a greater range and number of software artifacts.

Construct Validity.

The goal of cost-cognizant prioritization is to maximize the rate of "units-of-fault-cost-detected-per-unit-test-cost", and $APFD_C$ is designed to represent it. However, $APFD_C$ is not the only possible measure. For example, $APFD_C$ assigns no value to the subsequent tests that detect a fault already detected; such inputs may, however, help debuggers isolate the fault, and for that reason might be worth measuring. Further, the $APFD_C$ metric only partially captures the aspects of the effectiveness of prioritization. One might not even want to measure rate of detection; one might instead measure the percentage of tests in a prioritized test suite that must be run before all faults have been detected. Ultimately, we will need to also consider other measures for the purposes of assessing effectiveness.

Our measures also ignore the human costs that can be involved in maintaining and executing test suites. Our measures do not consider the analysis time required to prioritize tests. Previous work[14, 15, 16] has shown, however, that for the techniques considered, analysis time is much smaller than test execu-

tion time, or analysis can be performed automatically in off-hours prior to the critical regression testing period.

Conclusion Validity.

The number of programs and versions we considered was large enough to show differences among some of the cost-cognizant prioritization techniques we studied, but not for others. Although the use of more versions would have increased the power of the experiment, the cost of preparing each version is large, limiting our ability to make additional observations.

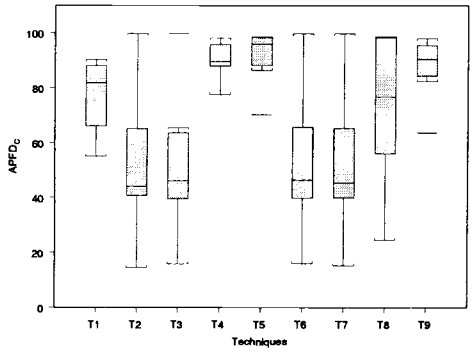
4.6 Data and Analysis

We now analyze the data for each of our research questions in turn. We first consider RQ1 through RQ5, for fault cost algorithm 1 only.

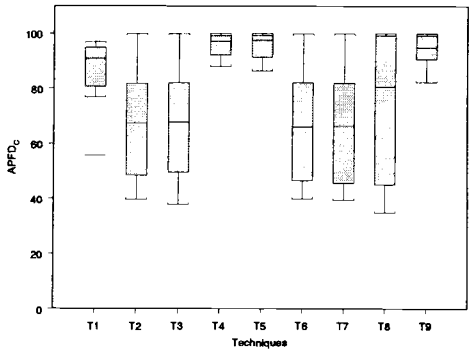
RQ1, RQ2. Our first two research questions consider whether cost-cognizant prioritization improves $APFD_C$ and whether $APFD_C$ values differ as *fault-hardness* varies. It is convenient to consider these questions together.

The boxplots in Figure 4.1 illustrate the $APFD_C$ values of the nine categories of prioritized test suites across all programs for each category of fault-hardness. T1 is the control group, i.e, the random prioritization technique. Examining the boxplots of the other prioritization techniques, T2 through T9, it seems that $APFD_C$ values do differ across techniques, at each hardness level, but the difference decreases as faults become easier to detect.

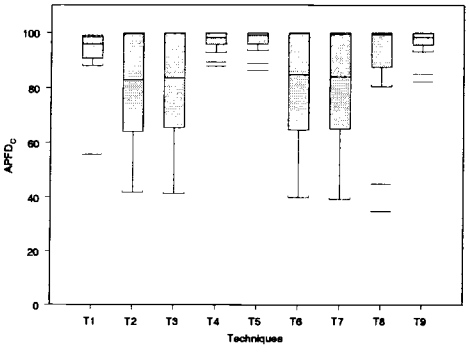
Using the S-PLUS 2000 statistical package to perform an ANOVA analysis, we were able to reject the null hypothesis that the $APFD_C$ means for various techniques are equal at each hardness level, confirming our boxplot observations.



(a) Hard faults.



(b) Moderate faults.



(c) Easy faults.

Code	Mnemonic
T1	random
T2	tcov–stat
T3	tcov–func
T4	acov–stat
T5	acov–func
T6	fi–tcov–func–bdiff
T7	fi–tcov–stat–bdiff
T8	fi–acov–func–bdiff
T9	fi–acov–stat–bdiff

(d) Prioritization techniques.

FIGURE 4.1: APFD$_C$ values across all programs for different fault hardnesses.

Also ANOVA analysis across each hardness level reveals that the $APFD_C$ means across different hardness levels are significantly different.

Table 4.4 shows the mean $APFD_C$ values for all techniques across all the fault hardness levels for all programs. Examining the table, we see that techniques T4, T5, and T9 always performed better than T1 (random) across all fault hardness levels. Hence there is improvement in the rate of fault detection for these techniques in comparison to random prioritization. Moreover, it is evident that the improvement over random prioritization decreases as the faults become easier to detect. For example, in the case of hard faults the mean $APFD_C$ value of the best technique, T5 (acov-func), was 14.61% greater than the $APFD_C$ for technique T1 (random); in the case of moderate faults, the $APFD_C$ for T5 was only 8.77% greater than that for T1; in the case of easy faults, the $APFD_C$ for T5 was only 4.65% greater. Also it is evident from the table that many techniques did not outperform the random technique (T1). Moreover, examining the boxplots, we see that there much wider variance in the performance of techniques T2, T3, T6, T7, and T8 as compared to techniques T4, T5, and T9. Hence the performance of techniques T2, T3, T6, T7, and T8 may be less predictable that of T4, T5, and T9.

RQ3. Our third research question considers whether incorporating feedback information into prioritization can improve $APFD_C$. From our results regarding research question RQ2, it is evident that cost-cognizant prioritization has greater applicability to the case of hard faults than to the case of moderate or easy faults. Therefore we will be considering $APFD_C$ values of prioritization techniques only on hard faults to address this and the following research questions.

Code	Name	Hard Fault Mean	Moderate Fault Mean	Easy Fault Mean
T5	addtl. cov. func.	91.72	95.48	97.14
T4	addtl. cov. stat.	90.26	95.14	96.89
T9	addtl. cov. stat. bdiff.	87.74	94.00	96.42
T1	random.	77.11	86.71	92.59
T8	addtl. cov. func. bdiff.	73.21	77.08	88.88
T6	total cov. func. bdiff.	52.43	67.11	80.23
T7	total cov. stat. bdiff.	51.97	66.61	79.83
T3	total cov. func.	51.74	67.15	80.09
T2	total cov. stat.	51.74	66.73	79.56

TABLE 4.4: Mean $APFD_C$ Values Across all Programs for Different Levels of Fault Hardness, Ranked by Hard Fault Mean.

Initial observation of our data led us to hypothesize that feedback information has an effect on $APFD_C$ values. This is suggested by looking at the mean $APFD_C$ values in Table 4.4, and the tables in Appendix, where in most of the cases, $APFD_C$ values of techniques incorporating feedback information were better than those for techniques that do not incorporate feedback information.

Figure 4.2 depicts the results more clearly; it shows the effects of incorporating feedback in each technique on the $APFD_C$ value. The vertical axis represents $APFD_C$ values for all the techniques incorporating feedback information. The horizontal axis represents $APFD_C$ values for all the techniques not incorporating feedback information. Each point in the graph represents the $APFD_C$ value for a technique with feedback versus $APFD_C$ for its counterpart without feedback for a particular program and version. Examining the figure, we see that almost 90% of the points lie above the x=y line, i.e, favoring the additional prioritization techniques. Moreover, looking at the figure, we see that

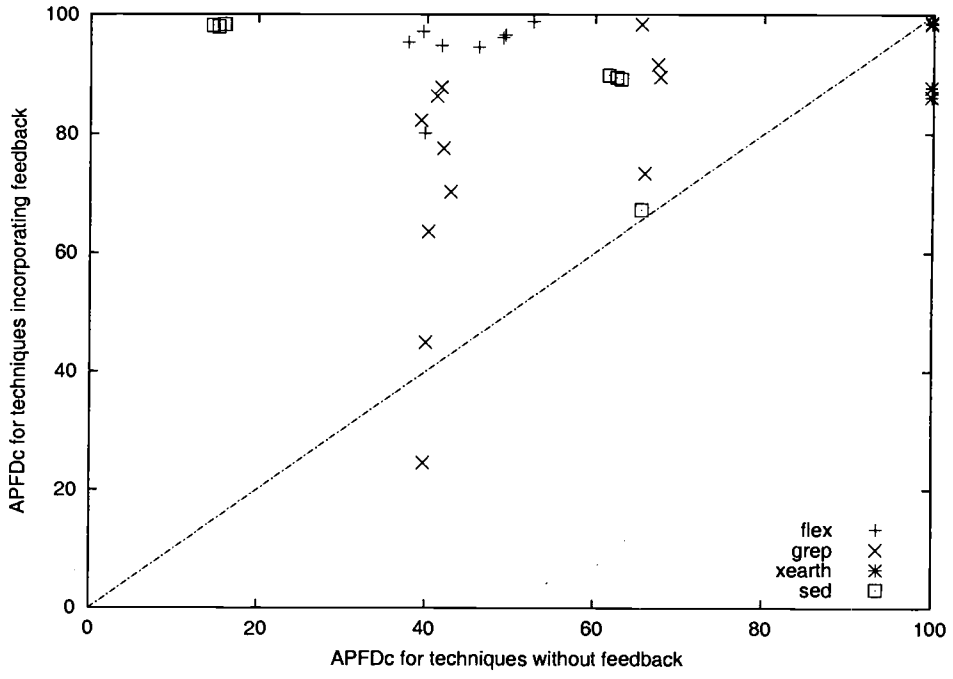


FIGURE 4.2: Feedback vs non-feedback techniques.

the cases favoring the additional prioritization techniques tend to have much higher $APFD_C$ values than their non-feedback counterparts, whereas the cases favoring the total prioritization techniques have $APFD_C$ values relatively close to those of their feedback-incorporating counterparts.

RQ4. Our fourth research question concerns the relationship between fine and coarse granularity prioritization techniques. Initial observation of the data led us to hypothesize that granularity has an effect on $APFD_C$ values. This is illustrated by looking at Table 4.4: for some cases the mean $APFD_C$ values of function level techniques are better than the mean $APFD_C$ values of corresponding statement level techniques while in some cases the mean $APFD_C$

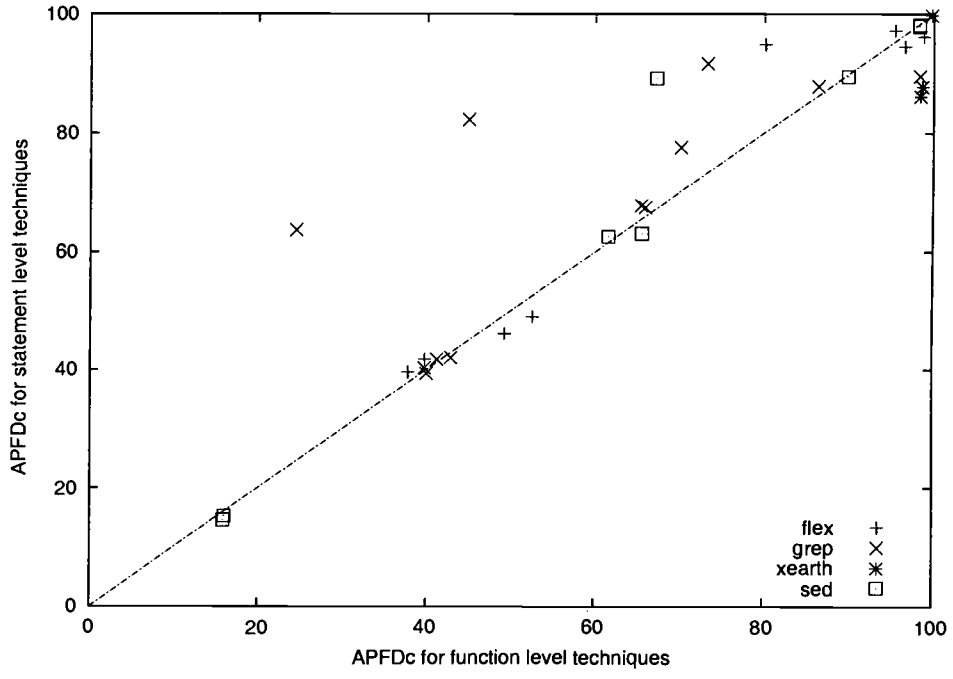


FIGURE 4.3: Statement level vs function level techniques.

values of statement level techniques are better than the mean $APFD_C$ values of corresponding function level techniques.

Figure 4.3 further investigates this research question. The vertical axis represents $APFD_C$ values for all the techniques operating at the statement level. The horizontal axis represents $APFD_C$ values for all the techniques operating at the function level. Each point in the graph represents the $APFD_C$ value for a technique operating at statement level versus the $APFD_C$ value for its corresponding function level technique for a particular program and version. Examining the figure, we see that almost 50% of the points lie above the x=y line, i.e, favoring statement level prioritization techniques, and the rest of the points lie below the x=y line, i.e, favoring function level prioritization techniques. Also looking at the figure, we see that almost 70% of the points lie

very near to the x=y line, indicating that for a majority of the cases considered, statement level and function level techniques performed nearly the same. However, looking at the figure, we also see that the cases favoring statement level techniques have much higher $APFD_C$ values than corresponding function level techniques more often than the reverse. In other words, statement level techniques had greater likelihood of providing a substantial improvement over function level than the reverse.

RQ5. Our fifth research question considers whether predictors of fault proneness can be used to improve $APFD_C$ values. We hypothesized that incorporation of such predictors would increase technique effectiveness. Surprisingly, looking at the tables in Appendix, we find that the $APFD_C$ values for the majority of techniques incorporating fault proneness information were worse than those for techniques without fault proneness.

Figure 4.4 further investigates this research question. The vertical axis represents the $APFD_C$ values for all the techniques incorporating fault proneness predictors. The horizontal axis represents $APFD_C$ values for all the techniques without fault proneness predictors. Each point in the graph represents the $APFD_C$ value for a technique with fault proneness predictors versus the $APFD_C$ value for its corresponding technique without fault proneness predictors for a particular program and version. Examining the figure, we see that almost 55% of the points lie below the x=y line, favoring the techniques without fault proneness predictors, and the rest of the points lie above the x=y line. Also looking at the figure, we see that almost 80% of the points lie near the x=y line. Morever, we also see that the cases favoring techniques without predictors of fault proneness have much higher $APFD_C$ values than the corresponding techniques

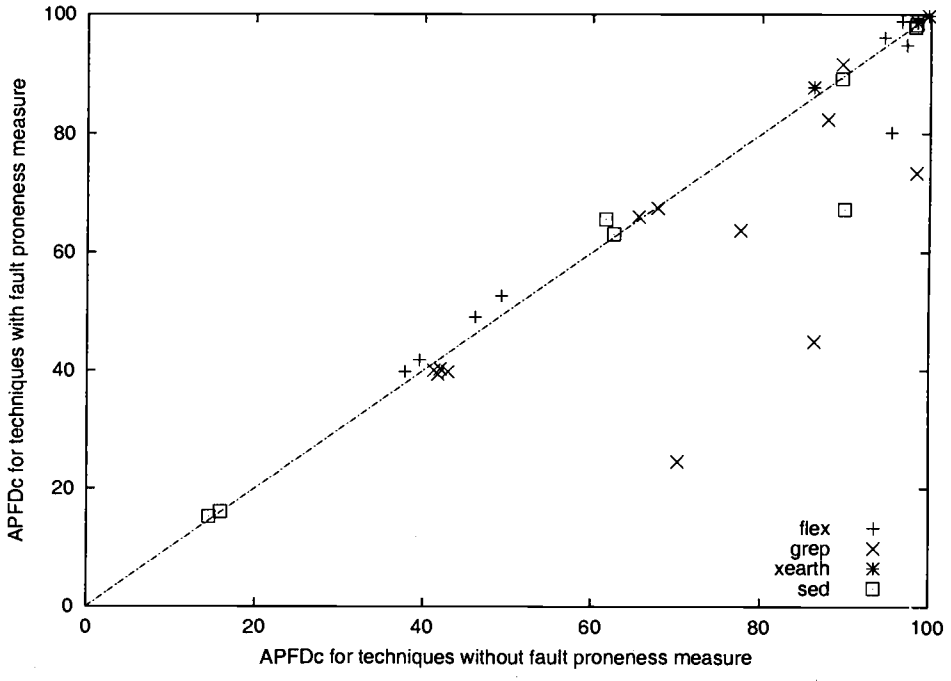


FIGURE 4.4: Fault proneness vs non-fault-proneness techniques.

with predictors of fault proneness more often than the reverse. In other words, techniques without any predictors of fault proneness had a greater likelihood of providing a substantial improvement than techniques with predictors of fault proneness.

RQ6. Our final research question considers how fault cost algorithm 1 compared to fault cost algorithm 2 in terms of its support for prioritization. Initial observation of our data led us to hypothesize that different fault cost algorithms could have an effect on $APFD_C$ values. This is suggested by looking at the tables in Appendix where, in some of the cases, $APFD_C$ values of techniques applied to programs where fault cost was calculated using fault cost algorithm

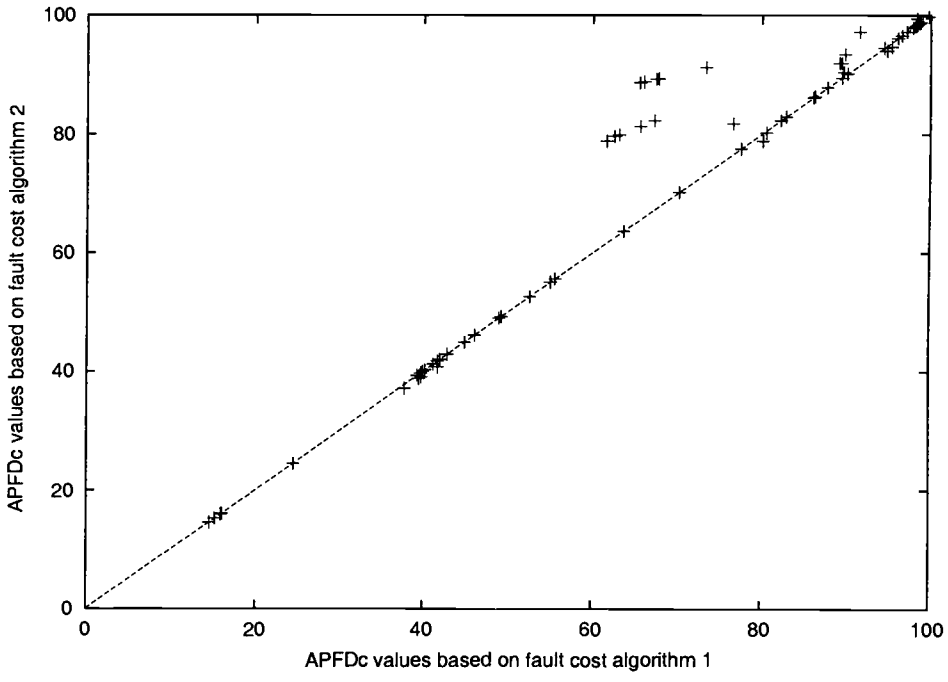


FIGURE 4.5: $APFD_C$ values for fault costs based on algorithm 2 vs fault costs based on algorithm 1.

2 were higher than $APFD_C$ values for techniques that were applied when fault cost was assessed using fault cost algorithm 1.

In Figure 4.5 the vertical axis represents the $APFD_C$ values for all the techniques when fault cost is calculated based on algorithm 2. The horizontal axis represents the $APFD_C$ values for all the techniques when fault cost is calculated based on algorithm 1. Each point in the graph represents the $APFD_C$ value for a technique having fault cost calculated based on fault cost algorithm 1 versus the $APFD_C$ value for its corresponding technique with fault cost being assessed based on fault cost algorithm 2 for a particular program and version. Examining the figure, we see that almost 22% of the points lie above the x=y line and the rest of the points lie either on the x=y line or very close to it. In other words,

we can say that most of the techniques applied to programs having fault cost based on algorithm 2 have equal or better $APFD_C$ values than the techniques applied to programs having fault cost based on algorithm 1.

CHAPTER 5

DISCUSSION

Keeping in mind the threats to validity for the empirical study presented in Chapter 4, our data and analysis provide several insights into the effectiveness of the cost-cognizant prioritization techniques that we examined. We now discuss these insights and their possible implications for practical application of cost-cognizant test case prioritization.

Of greatest practical significance, our data and analysis indicates that some of the cost cognizant prioritization techniques can substantially improve the rate of fault detection, particularly when faults are harder to detect. Additional statement coverage, additional function coverage, and additional binary diff statement coverage all performed better than random ordering. This was true for all fault hardness levels, but with much greater potential impact on hard faults.

Also in our study, prioritization techniques with feedback information outperformed techniques without any feedback information. Moreover, the techniques with feedback had smaller variance in results compared to techniques without feedback. Hence there would seem to be more likelihood of obtaining predictable results using techniques that incorporate feedback.

Also, the fact that, in most cases, performance was similar for both function level and statement level techniques is important. The coarser analysis used by function level techniques renders them less costly, and less intrusive than

statement level techniques. Our results indicate that, on average, function level techniques were often similar in effectiveness to statement level techniques and thus, there could be benefits to using them.

Again considering overall results, techniques operating without any feedback information always performed worse than random techniques. This fact was quite surprising given our initial intution and previous studies [16] where techniques without feedback (e.g. total coverage) often performed better than random. One possible explanation is that faults in our subjects might be located on obscure control flow paths or revealed by test cases with very low coverage. Hence total coverage prioritization will schedule the test cases with low coverage very late in the prioritized order, while a random technique has more likelihood of scheduling them earlier in the prioritized order.

Our investigation of incorporation of measures of fault proneness into prioritization showed that they performed worse than corresponding non-fault-proneness techniques which was also quite surprising. One possible explanation for this behavior is that there is very little correlation between the changed code and fault distribution for our subjects. For example, faults might not be uniformly distributed over the changed functions. Suppose there is a program with ten functions, fn1 through fn10, out of which six functions are changed, fn1 through fn6. Suppose the program has a single fault f1 and a test suite of three test cases, t1 through t3. Suppose Table 5.1 illustrates the functions covered and faults revealed by these test cases. In this case, the additional functional coverage technique will schedule the test cases in order t3–t2–t1, while additional binary diff function coverage will place the test cases in order t1–t2–t3. Hence for this example, the technique that omits fault proneness will perform better than the corresponding technique using fault proneness measures.

Test	Function covered	Fault revealed
t1	fn1, fn2, fn3	-
t2	fn4, fn5	-
t3	fn6, fn7, fn8, fn9, fn10	f1

TABLE 5.1: Fault Proneness Example.

The fault proneness techniques in general depend on an assumed correlation between changes and faults. When this correlation does not hold, these techniques may not perform well. Before using such techniques, practitioners need to know about the likely distribution of faults relative to changes in their systems.

Finally, we observed that when fault costs are assessed with algorithm 2, techniques have either the same or better $APFD_C$ values than when fault costs are calculated based on algorithm 1. Hence there could be benefits to using fault cost algorithm 2 for calculating fault costs.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this thesis, we have investigated cost-cognizant prioritization. We have described methods for assigning fault costs and test costs. We have further described several cost-cognizant prioritization techniques for regression testing, and emprically examined their relative abilities to improve "units-of-fault-cost-detected-per-unit-test-cost" during regression testing. The results of our study indicate that cost-cognizant prioritization can substantially improve the rate of fault detection of test suites, and expose several tradeoffs between prioritization techniques.

The results of our study also suggest several avenues for future work.

First, there is a need to perform additional studies using other programs and types of test suites, and a wider range and distribution of hard to detect faults, to help generalize our empirical results.

Second, it may be useful to investigate alternative techniques. One such alternative involves taking some other mathematical combinations of test cost and test criticality while prioritizing.

Third, assessing fault cost based on frequency of usage is only one heuristic. Alternative heuristics may be possible and preferable in different situations, and such heuristics should be investigated.

Fourth, our analysis revealed that techniques not using fault proneness information performed better than techniques using fault proneness information.

This surprising behavior should be further investigated; perhaps some other heuristic can improve upon the binary difference based techniques.

Fifth, at present we did not take into account the fault revealing data associated with each test run on previous versions of the software. It would be interesting to see how prioritization will perform if we also consider such data while prioritizing for the next version.

Sixth, it would be interesting to study prioritization techniques that are hybrids of some of the techniques described in Chapter 3. For example, we could first sort tests into various classes based on the fault-proneness, and then among those classes we can apply other techniques like additional statement coverage to order the tests within classes.

Seventh, differences in the performance of the various prioritization techniques we investigated mandate further study of the factors that underlie the relative effectiveness of various techniques. A desirable outcome of such study would be techniques for predicting, for particular programs, types of test suites, and classes of modifications, which prioritization technique would be more cost effective.

Eighth, it would be important to study the cost benefits of the techniques in practical environments. $APFD_C$ only captures the degree of effectiveness of various techniques, however, it does not represent the actual cost savings for an organization using prioritization techniques. It would be interesting to investigate various cost models that would help to assess the cost benefits of techniques, in practical environments.

Finally, the test case prioritization problem has many more facets than we have considered here. The test case prioritization problem that we have examined here is cost-cognizant prioritization for rate of fault cost detection; other

objectives listed in Chapter 2 are also of interest. Further, it would be interesting to consider applying prioritization in conjunction with test selection and minimization.

Through the results reported in this thesis, and future work in these areas, we hope to provide software practitioners with cost-effective techniques for improving testing and regression testing process through prioritization of tests.

BIBLIOGRAPHY

[1] T. Ball. On the limit of control flow analysis for regression test selection. In *Proceedings of the ACM International Symposium on Software Testing and Analysis*, pages 134–142, March 1998.

[2] L. C. Briand, J. Wust, S. Ikonomovski, and H. Lounis. Investigating quality factors in object oreinted designs: an industrial case study. In *Proceedings of the International Conference on Software Engineering*, pages 345–354, May 1999.

[3] V. Chen, D. S. Rosenblum, and K. Vo. TestTube: a system for selective regression testing. In *Proceedings of the 16th International Conference on Software Engineering*, pages 211–220, May 1994.

[4] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 329–338, May 2001.

[5] S. Elbaum, A. Malishevsky, and G. Rothermel. Test Case Prioritization: A Family of Empirical Studies. Technical Report TR: 01-60-08, Oregon State University, May 2001.

[6] M. J. Harrold and G. Rothermel. Aristotle: A system for research on and development of program analysis based tools. Technical Report OSU-CISRC- 3/97-TR17, Ohio State University, March 1997.

[7] F. Lanubile, A. Lonigro, and G. Visaggio. Comparing models for identifying fault-prone software components. In *Proceedings of the 7th International Conference on Software Engineering and Knowledge Engineering*, pages 12–19, June 1995.

[8] J.C. Munson and T.M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Trans. on Softw. Eng.*, pages 423–433, May 1992.

[9] J. D. Musa. Operational profiles in software reliability engineering. *IEEE Software*, 10(2):14–32, March 1993.

[10] J. D. Musa. *Software Reliability Engineering.* McGraw-Hill, New York, NY, 1999.

[11] K. Onoma, W-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Communications of the ACM*, 41(5):81–86, May 1988.

[12] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686.

[13] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, August 1996.

[14] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[15] 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 *Proceedings of the International Conference on Software Maintenance*, pages 34–43, November 1998.

[16] G. Rothermel, R. H. Untch, C. Chengyun, and M. J. Harrold. Test case prioritization. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.

[17] G. Schulmeyer and J. McManus. *Handbook of Software Quality Assurance.* Prentice Hall, New York, NY, 3rd edition, 1999.

[18] W. E. Wong, R. J. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. *Software - Practice and Experience*, 28(4):347–369, April 1998.

[19] W. E. Wong, R. J. Horgan, A. Mathur, and A. Pasquini. Test set size minimization and fault detection effectiveness: A case study in space application. In *Proceedings of the 21st Annual International Computer Software & Applications Conference*, pages 522–528, August 1997.

APPENDIX

TABLES GIVING APFD$_C$ ACROSS EACH PROGRAM.

Technique	V1	V2	V3	V4
rand	55.60	94.19	89.74	–
tcov-stat	41.72	41.99	67.74	–
tcov-func	41.22	42.88	65.51	–
acov-stat	87.87	77.55	89.54	–
acov-func	86.37	70.21	98.38	–
fi-tcov-func-bdiff	39.98	39.74	66.00	–
fi-tcov-stat-bdiff	39.33	40.25	67.47	–
fi-acov-func-bdiff	44.91	24.51	73.33	–
fi-acov-stat-bdiff	82.32	63.65	91.66	–

(a) Grep

Technique	V2	V3
rand	55.07	–
tcov-stat	99.74	–
tcov-func	99.74	–
acov-stat	87.77	–
acov-func	98.66	–
fi-tcov-func-bdiff	99.74	–
fi-tcov-stat-bdiff	99.74	–
fi-acov-func-bdiff	98.43	–
fi-acov-stat-bdiff	86.17	–

(b) Xearth

Technique	V1	V2	V3	V4
rand	–	82.94	80.58	–
tcov-stat	–	46.13	39.51	–
tcov-func	–	49.26	37.79	–
acov-stat	–	94.53	97.18	–
acov-func	–	96.59	95.40	–
fi-tcov-func-bdiff	–	52.60	39.80	–
fi-tcov-stat-bdiff	–	49.00	41.73	–
fi-acov-func-bdiff	–	98.82	80.14	–
fi-acov-stat-bdiff	–	96.14	94.86	–

(c) Flex

Technique	V2	V3
rand	76.61	86.22
tcov-stat	62.56	14.54
tcov-func	61.63	15.90
acov-stat	89.49	98.16
acov-func	89.88	98.29
fi-tcov-func-bdiff	65.58	16.03
fi-tcov-stat-bdiff	63.09	15.22
fi-acov-func-bdiff	67.25	98.29
fi-acov-stat-bdiff	89.23	97.92

(d) Sed

TABLE A.1: APFD$_C$ Values Across all Programs for Hard Faults with Fault Cost Algorithm 1.

Technique	V1	V2	V3	V4
rand	55.60	94.19	90.42	–
tcov-stat	41.72	41.99	89.32	–
tcov-func	41.22	42.88	88.64	–
acov-stat	87.87	77.55	89.54	–
acov-func	86.37	70.21	99.43	–
fi-tcov-func-bdiff	39.98	39.74	88.80	–
fi-tcov-stat-bdiff	39.33	40.25	89.28	–
fi-acov-func-bdiff	44.91	24.51	91.21	–
fi-acov-stat-bdiff	82.32	63.65	97.23	–

(a) Grep

Technique	V2	V3
rand	55.07	–
tcov-stat	99.74	–
tcov-func	99.74	–
acov-stat	87.77	–
acov-func	98.66	–
fi-tcov-func-bdiff	99.74	–
fi-tcov-stat-bdiff	99.74	–
fi-acov-func-bdiff	98.43	–
fi-acov-stat-bdiff	86.17	–

(b) Xearth

Technique	V1	V2	V3	V4
rand	–	82.94	80.20	–
tcov-stat	–	46.13	38.79	–
tcov-func	–	49.26	37.15	–
acov-stat	–	94.53	97.22	–
acov-func	–	96.59	94.69	–
fi-tcov-func-bdiff	–	52.60	39.06	–
fi-tcov-stat-bdiff	–	49.00	40.75	–
fi-acov-func-bdiff	–	98.82	78.86	–
fi-acov-stat-bdiff	–	96.14	94.01	–

(c) Flex

Technique	V2	V3
rand	81.84	86.22
tcov-stat	79.65	14.54
tcov-func	78.86	15.90
acov-stat	91.95	98.16
acov-func	93.42	98.29
fi-tcov-func-bdiff	81.31	16.03
fi-tcov-stat-bdiff	79.90	15.22
fi-acov-func-bdiff	82.29	98.29
fi-acov-stat-bdiff	91.99	97.92

(d) Sed

TABLE A.2: $APFD_C$ Values Across all Programs for Hard Faults with Fault Cost Algorithm 2.

Technique	V1	V2	V3	V4
rand	55.60	94.40	90.81	94.79
tcov-stat	41.72	49.87	67.31	60.75
tcov-func	41.22	51.67	67.69	63.21
acov-stat	87.87	92.78	89.33	99.21
acov-func	86.37	88.87	97.47	99.13
fi-tcov-func-bdiff	39.98	49.99	66.11	63.11
fi-tcov-stat-bdiff	39.33	50.58	66.40	60.59
fi-acov-func-bdiff	44.91	35.05	80.60	99.25
fi-acov-stat-bdiff	82.32	85.12	93.75	99.40

(a) Grep

Technique	V2	V3
rand	76.80	96.96
tcov-stat	99.44	99.88
tcov-func	99.43	99.87
acov-stat	93.64	99.99
acov-func	99.32	99.99
fi-tcov-func-bdiff	99.43	99.86
fi-tcov-stat-bdiff	99.44	99.88
fi-acov-func-bdiff	99.25	99.99
fi-acov-stat-bdiff	92.94	99.99

(b) Xearth

Technique	V1	V2	V3	V4
rand	94.74	89.82	80.58	–
tcov-stat	48.37	71.21	39.51	–
tcov-func	49.56	73.32	37.79	–
acov-stat	99.18	97.29	97.18	–
acov-func	99.32	94.20	95.40	–
fi-tcov-func-bdiff	46.44	75.79	39.80	–
fi-tcov-stat-bdiff	45.48	73.14	41.73	–
fi-acov-func-bdiff	34.70	97.81	80.14	–
fi-acov-stat-bdiff	99.30	97.28	94.86	–

(c) Flex

Technique	V2	V3
rand	83.59	95.76
tcov-stat	74.17	81.85
tcov-func	72.83	82.09
acov-stat	92.17	97.90
acov-func	91.25	99.00
fi-tcov-func-bdiff	75.61	82.11
fi-tcov-stat-bdiff	74.27	81.91
fi-acov-func-bdiff	77.79	99.00
fi-acov-stat-bdiff	90.63	97.91

(d) Sed

TABLE A.3: $APFD_C$ Values Across all Programs for Moderate Faults with Fault Cost Algorithm 1.

Technique	V1	V2	V3	V4
rand	55.60	95.69	91.22	94.79
tcov-stat	41.72	51.81	83.33	60.75
tcov-func	41.22	53.92	84.64	63.21
acov-stat	87.87	95.51	89.30	99.21
acov-func	86.37	92.22	98.33	99.13
fi-tcov-func-bdiff	39.98	51.48	84.821	63.11
fi-tcov-stat-bdiff	39.33	52.91	84.70	60.59
fi-acov-func-bdiff	44.91	42.48	91.74	99.25
fi-acov-stat-bdiff	82.32	94.76	97.86	99.40

(a) Grep

Technique	V2	V3
rand	76.80	96.96
tcov-stat	99.44	99.88
tcov-func	99.43	99.87
acov-stat	93.64	99.99
acov-func	99.32	99.99
fi-tcov-func-bdiff	99.43	99.86
fi-tcov-stat-bdiff	99.44	99.88
fi-acov-func-bdiff	99.25	99.99
fi-acov-stat-bdiff	92.94	99.99

(b) Xearth

Technique	V1	V2	V3	V4
rand	94.74	92.75	80.20	–
tcov-stat	48.37	85.74	38.79	–
tcov-func	49.56	88.65	37.15	–
acov-stat	99.18	98.92	97.22	–
acov-func	99.32	91.64	94.69	–
fi-tcov-func-bdiff	46.44	90.48	39.06	–
fi-tcov-stat-bdiff	45.48	88.72	40.75	–
fi-acov-func-bdiff	34.70	98.16	78.86	–
fi-acov-stat-bdiff	99.30	99.01	94.01	–

(c) Flex

Technique	V2	V3
rand	90.38	97.85
tcov-stat	89.07	96.18
tcov-func	87.50	96.20
acov-stat	94.97	97.70
acov-func	93.67	99.16
fi-tcov-func-bdiff	88.87	96.20
fi-tcov-stat-bdiff	88.75	96.13
fi-acov-func-bdiff	91.11	99.16
fi-acov-stat-bdiff	92.71	97.85

(d) Sed

TABLE A.4: $APFD_C$ Values Across all Programs for Moderate Faults with Fault Cost Algorithm 2.

Technique	V1	V2	V3	V4
rand	55.60	94.40	90.81	94.79
tcov-stat	41.72	49.87	67.31	60.75
tcov-func	41.22	51.67	67.69	63.21
acov-stat	87.87	92.78	89.33	99.21
acov-func	86.37	88.87	97.47	99.13
fi-tcov-func-bdiff	39.98	49.99	66.11	63.11
fi-tcov-stat-bdiff	39.33	50.58	66.40	60.59
fi-acov-func-bdiff	44.91	35.05	80.60	99.25
fi-acov-stat-bdiff	82.32	85.12	93.75	99.40

(a) Grep

Technique	V2	V3
rand	90.51	98.21
tcov-stat	99.80	99.93
tcov-func	99.79	99.93
acov-stat	97.76	99.99
acov-func	99.73	99.99
fi-tcov-func-bdiff	99.79	99.92
fi-tcov-stat-bdiff	99.80	99.94
fi-acov-func-bdiff	99.73	99.96
fi-acov-stat-bdiff	97.51	99.99

(b) Xearth

Technique	V1	V2	V3	V4
rand	97.91	89.82	90.81	98.79
tcov-stat	84.48	71.21	60.57	99.82
tcov-func	86.80	73.32	59.61	99.82
acov-stat	98.05	97.29	98.35	97.63
acov-func	98.67	94.20	97.72	99.63
fi-tcov-func-bdiff	87.57	75.79	61.90	99.82
fi-tcov-stat-bdiff	86.76	73.14	63.77	99.82
fi-acov-func-bdiff	91.69	97.81	91.20	99.73
fi-acov-stat-bdiff	99.75	97.28	97.45	98.39

(c) Flex

Technique	V2	V3
rand	88.04	96.86
tcov-stat	81.29	86.96
tcov-func	80.32	87.16
acov-stat	94.13	98.46
acov-func	93.49	99.24
fi-tcov-func-bdiff	82.34	87.18
fi-tcov-stat-bdiff	81.37	87.03
fi-acov-func-bdiff	83.92	99.24
fi-acov-stat-bdiff	93.21	98.39

(d) Sed

Technique	V1	V2	V3	V4
rand	98.64	98.02	99.00	98.99
tcov-stat	71.01	99.27	99.99	99.10
tcov-func	71.73	99.73	99.99	99.50
acov-stat	99.88	99.84	99.90	99.92
acov-func	99.88	99.94	99.96	99.96
fi-tcov-func-bdiff	71.40	99.40	99.99	99.50
fi-tcov-stat-bdiff	71.27	98.71	99.96	98.96
fi-acov-func-bdiff	99.88	99.90	99.96	99.92
fi-acov-stat-bdiff	99.88	99.94	99.96	99.92

(e) Make

TABLE A.5: $APFD_C$ Values Across all Programs for Easy Faults with Fault Cost Algorithm 1.

Technique	V1	V2	V3	V4
rand	55.60	95.69	91.22	94.79
tcov-stat	41.72	51.81	83.33	60.75
tcov-func	41.22	53.92	84.64	63.21
acov-stat	87.87	95.51	89.30	99.21
acov-func	86.37	92.22	98.33	99.13
fi-tcov-func-bdiff	39.98	52.26	83.00	63.11
fi-tcov-stat-bdiff	39.33	52.85	82.65	60.59
fi-acov-func-bdiff	44.91	37.39	93.23	99.25
fi-acov-stat-bdiff	82.32	88.97	97.70	99.40

(a) Grep

Technique	V2	V3
rand	97.13	99.22
tcov-stat	99.97	99.98
tcov-func	99.97	99.98
acov-stat	99.73	99.99
acov-func	99.94	99.99
fi-tcov-func-bdiff	99.97	99.98
fi-tcov-stat-bdiff	99.97	99.98
fi-acov-func-bdiff	99.96	99.94
fi-acov-stat-bdiff	97.70	99.99

(b) Xearth

Technique	V1	V2	V3	V4
rand	98.24	92.75	97.88	98.79
tcov-stat	88.57	85.74	94.01	99.82
tcov-func	91.18	88.65	95.39	99.82
acov-stat	98.20	98.92	99.68	97.63
acov-func	98.78	91.64	99.62	99.63
fi-tcov-func-bdiff	92.28	90.48	96.60	99.82
fi-tcov-stat-bdiff	91.52	88.72	96.75	99.82
fi-acov-func-bdiff	98.13	98.16	99.06	99.73
fi-acov-stat-bdiff	99.80	99.01	99.58	98.39

(c) Flex

Technique	V2	V3
rand	98.03	98.92
tcov-stat	98.00	98.28
tcov-func	97.72	98.35
acov-stat	98.50	98.99
acov-func	98.34	99.57
fi-tcov-func-bdiff	98.01	98.35
fi-tcov-stat-bdiff	97.99	98.32
fi-acov-func-bdiff	98.40	99.57
fi-acov-stat-bdiff	98.69	98.90

(d) Sed

Technique	V1	V2	V3	V4
rand	98.64	98.02	99.00	98.99
tcov-stat	71.01	99.27	99.99	99.10
tcov-func	71.73	99.73	99.99	99.50
acov-stat	99.88	99.84	99.90	99.92
acov-func	99.88	99.94	99.96	99.96
fi-tcov-func-bdiff	71.40	99.40	99.99	99.50
fi-tcov-stat-bdiff	71.27	98.71	99.96	98.96
fi-acov-func-bdiff	99.88	99.90	99.96	99.92
fi-acov-stat-bdiff	99.88	99.94	99.96	99.92

(e) Make

TABLE A.6: $APFD_C$ Values Across all Programs for Easy Faults with Fault Cost Algorithm 2.