AN ABSTRACT OF THE DISSERTATION OF

Alexey Grigorievich Malishevsky for the degree of Doctor of Philosophy in Computer Science presented on June 19, 2003.

Title: Test Case Prioritization.

Abstract approved: Redacted for privacy

Gregg E. Rothermel

Regression testing is an expensive software engineering activity intended to provide confidence that modifications to a software system have not introduced faults. Test case prioritization techniques help to reduce regression testing cost by ordering test cases in a way that better achieves testing objectives. In this thesis, we are interested in prioritizing to maximize a test suite's rate of fault detection, measured by a metric, APFD, trying to detect regression faults as early as possible during testing.

In previous work, several prioritization techniques using low-level code coverage information had been developed. These techniques try to maximize APFD over a sequence of software releases, not targeting a particular release. These techniques' effectiveness was empirically evaluated.

We present a larger set of prioritization techniques that use information at arbitrary granularity levels and incorporate modification information, targeting prioritization at a particular software release. Our empirical studies show significant improvements in the rate of fault detection over randomly ordered test suites.

Previous work on prioritization assumed uniform test costs and fault severities, which might not be realistic in many practical cases. We present a new cost-cognizant metric, $APFD_C$, and prioritization techniques, together with approaches for measuring and estimating these costs. Our empirical studies evaluate prioritization in a cost-cognizant environment.

Prioritization techniques have been developed independently with little consideration of their similarities. We present a general prioritization framework that allows us to express existing prioritization techniques by a framework algorithm using parameters and specific functions.

Previous research assumed that prioritization was always beneficial if it improves the APFD metric. We introduce a prioritization cost-benefit model that more accurately captures relevant cost and benefit factors, and allows practitioners to assess whether it is economical to employ prioritization.

Prioritization effectiveness varies across programs, versions, and test suites. We empirically investigate several of these factors on substantial software systems and present a classification-tree-based predictor that can help select the most appropriate prioritization technique in advance.

Together, these results improve our understanding of test case prioritization and of the processes by which it is performed.

Test Case Prioritization

by
Alexey Grigorievich Malishevsky

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented June 19, 2003
Commencement June 2004

Doctor of Philosophy dissertation of Alexey Grigorievich Malishevsky presented on June 19, 2003

APPROVED:

Redacted for privacy

Major Professor, representing Computer Science

Redacted for privacy

Director of the School of Electrical Engineering and Computer Science

Redacted for privacy

Dean of the Graduate School

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

Redacted for privacy

Alexey Grigorievich Malishevsky, Author

ACKNOWLEDGMENTS

I give my biggest thanks to my adviser and mentor, Gregg Rothermel, who supported and guided me from the first conception of the topic to the very end. Because of him, I was able to participate in a variety of exciting experimental studies and had the privilege of being a part of a team that conducted a large number of research activities in an exciting area of regression testing. He guided the research activities and was the major contributor in interpretation of results and writing several articles we published on which a major portion of my work is based.

I give my special thanks to Sebastian Elbaum for his extensive help in experimental studies, especially in the area of data analysis. He devised and computed the code-based fault index metric that we used extensively. He contributed many significant ideas, analyzed data and interpreted the results for many experiments, organized and guided several empirical studies, made large contributions to many articles we published, and, together with my adviser, guided my research overall. In addition, he guided the development of Bash and Gzip experimental subjects. He supported several students who made significant contributions to this research.

I am very thankful to Margaret Burnett, Thomas Dietterich, Prasad Tadepalli, and David Sullivan for serving on my committee. They gave me valuable feedback on my work. I am especially thankful to Thomas Dietterich practical suggestions for improving this work and suggestions for future empirical studies.

My thanks go to Xuemei Qiu who contributed many ideas related to test suite composition experiments and collected portions of data for this study. She finalized preparation of the Empire subject and collected data related to this subject used by several empirical studies.

My thanks go to Amit Goel who has created infrastructure for several medium-size subjects and collected subjects-related data used in several studies. He and

Joseph Ruthruff contributed many ideas toward our studies of cost cognizance and conducted several experiments related to this concept.

I appreciate very much Praveen Kallakuri and Satya Kanduri who were major contributors for the empirical studies on factors affecting prioritization effectiveness. Praveen Kallakuri also made significant contributions to the test suite composition studies, and Satya Kanduri helped develop the classification tree-based predictor for prioritization effectiveness.

Much appreciation goes to Brian Davia for initially preparing the Empire experimental subject, and to Praveen Kallakuri for preparing the Bash experimental subject, which were extensively used in experiments. David Gable revised the techniques based on fault-proneness and provided the tools computing DIFF-based information. Thanks to Adam Ashenfelter, Sean Callan, Dan Chirica, Hyunsook Do, Desiree Dunn, David Gable, and Dalai Jin, who prepared a set of medium-sized experimental subjects.

Thanks to Siemens Corporate Research who provided us with a set of initial small-sized experimental subjects, and to Phyllis Frankl, Filip Vokolos, and Chengyun Chu, who provided the space subject.

I thank the National Science Foundation for supporting my research with grants.

I am greatly thankful to Oregon State University and the Computer Science Department for being my home for the last ten years and for giving me the opportunity to achieve my dreams.

I give the greatest thanks to my parents, Elena and Grigory, for emotional support through school years especially during difficult times. They made it all possible for me to achieve my objectives by guiding me and keeping my spirits high especially at rough times.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

LIST OF TABLES (Continued)

LIST OF TABLES (Continued)

LIST OF APPENDIX TABLES

Test Case Prioritization

CHAPTER 1

INTRODUCTION AND BACKGROUND

1.1 Introduction

Each time a software system is modified and is to be released, it is regression tested. Regression testing is similar to testing in general: it involves executing test cases and checking the results for correctness. Regression testing, however, is done to ensure that modifications to code have not created new faults (software has not regressed) or that modifications fulfilled their intended purpose by correctly altering software functionality.

During initial, *development testing*, software is validated for the first time as it is being written. Development testing is usually done only once during the software life-cycle. Thus, its cost is amortized over the software's lifetime. On the other hand, regression testing is done before each new software release as part of software maintenance. Being performed multiple times, regression testing can have a profound effect on the software budget. Because regression testing itself accounts for a large percentage of software cost [11, 63, 114, 117, 170], this means that even small reductions in regression testing cost can have a profound effect on the software cost. In addition, reducing regression testing time can reduce the time needed to create a new software version and allow engineers to release this version earlier than would be possible otherwise.

Frequently, a test suite is developed for regression testing, and reused across different regression testing sessions. To test new features, engineers add new test cases to this test suite; as a result, the suite grows and the cost of regression testing increases.

To reduce the cost of regression testing, several approaches have been developed. In regression test selection, only a subset of a test suite is selected and used to validate the software following changes [26, 116, 170]. In test suite reduction, at some point of time, the test suite is reduced, permanently discarding some test cases from the test suite [25, 73, 147].

In some cases, for example, when the software is safety critical, such as in flight control or medical equipment software; we cannot (or are not allowed to) omit test cases during regression testing. In this case, a different approach exists to reduce costs: test engineers may *prioritize* their regression test cases so that those which are more important, by some measure, are run earlier in the regression testing process.

Test case prioritization techniques 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 as quickly as possible, exercises features in order of frequency of use, or reflects their historically observed abilities to detect faults. In this thesis, we are concerned primarily with improving test suites' rate of fault detection. An improved rate of fault detection can provide earlier feedback on the system under test, enable earlier debugging, and increase the likelihood that, if testing is prematurely halted, those test cases that offer the greatest fault detection ability in the available testing time will have been executed.

Before we began this research, several prioritization techniques had been developed [174, 175, 178, 213]. These techniques had been explored and their effective-

ness evaluated in achieving certain goals. However, only limited studies of these techniques had been performed, using a few small programs. A larger pool of more realistic subjects has to be developed and more experiments performed in order to generalize results. This research addresses this need, by utilizing a larger number of more realistic program subjects for prioritization experiments.

Prior to this research, existing prioritization techniques did not take into account modifications made to the code, missing important information about the software. This research thus develops a variety of prioritization techniques utilizing information about code modifications in order to better predict the fault-revealing capability of test cases.

Previous prioritization techniques were also limited in their handling of test costs and fault severities, for the most part failing to account for differences in them.[1] While this could be acceptable in some situations, sometimes we cannot ignore these differences. Thus, this research develops prioritization techniques that utilize test cost and fault severity information, develops a metric to assess cost-cognizant prioritization effectiveness, and conducts a case study to explore cost cognizance.

Previously, researchers developed several prioritization techniques independently, providing separate algorithms for each. However, if we look closer, we can see that all of these previous techniques, as well as the new techniques we have created, have similarities. They evaluate each test case using some objective function

---

[1] Wong et al. [213] suggest the use of a coverage-based prioritization technique that uses test costs; however, they do not formally present such a technique or evaluate its effectiveness.

and order them using these functions. This research explores these commonalities and creates a unifying framework for prioritization techniques.

Previous studies of prioritization effectiveness used an objective metric to evaluate the quality of test orderings. However, even when this metric suggests a numeric advantage for one technique in comparison to another, this numeric advantage may not be correlated with meaningful cost savings in practice. Thus, this research develops a cost model with which engineers can assess whether a given prioritization technique is economical.

Previous research, and our own research, has also shown that prioritization can be affected by a wide variety of factors. However, no previous work has explored these factors. Thus, this research's empirical studies explore how several characteristics of a software system and its test cases affect prioritization. Knowledge about factors can facilitate the creation of a prediction mechanism for prioritization technique performance and create a set of guidelines for software developers who can target the software design and modifications so that prioritization can be successful.

Finally, given the variance we have observed in prioritization technique performance as factors vary, an important issue for the practical application of prioritization is the ability to predict which prioritization technique will be most beneficial under a given set of circumstances. This research shows how to use program and test case properties to better determine which prioritization technique will be the most beneficial given specific metrics.

Recently, researchers at Microsoft [185] have applied prioritization to test suites for several multi-million line software systems and found it highly efficient even on such large systems. This suggests that this problem is not merely academic: there is interest in prioritization for large-scale industrial applications.

1.2 Background and Previous Work

1.2.1 Testing

The purpose of software verification is to ensure that the software satisfies its specifications. Verification includes testing and analysis (informal analysis, code walkthroughs, code inspections, correctness proofs, and symbolic execution).

Testing involves, in general, executing the software and verifying that it behaves according to specification. Usually, in testing, we employ a test suite including sets of inputs (test cases) which are applied to the software to verify its behavior. The main difficulty in such testing is that software, in general, does not have the continuity property, meaning that small differences in inputs can cause dramatic differences in software behavior [63]. In most cases, testing itself can only show the presence of bugs, not their absence.

More formally, let P be a program, D be the input domain (set of allowable inputs) of P, and R be the range (set of program outputs given inputs from D) of P. P is *correct* for an input $d \in D$ (we call it a test case) iff its output $P(d)$ satisfies requirements. Program P is correct iff it is correct for every $d \in D$. If $P(d)$ is incorrect for some $d \in D$, we call this a *failure*. During its execution, an incorrect state that program P enters is called a fault. In order for a failure to occur, a fault must occur. We call a finite set of test cases T a *test suite*. If every test case $t \in T$ runs without exposing a failure, T is *successful* [63].

In principle, one way to test is to employ exhaustive testing by including a program's whole input domain in its test suite. While in some cases this is feasible (e.g., embedded systems with a small set of allowable finite inputs), it becomes impractical in most other cases. If inputs are of simple data types with limited size,

exhaustive test suites can be finite, but often unrealistically large. Thus, test suites must be created according to some rules.

A set of rules governing test suite composition is called a test adequacy criterion. A test adequacy criterion expresses conditions that a test suite must satisfy. Test adequacy criterion C is consistent iff for every two test suites T_1 and T_2 that satisfy C, they are both successful or both unsuccessful. Test adequacy criterion C is called complete if, when P is incorrect, there exists a test suite that is unsuccessful and belongs to C [63].

In practice, test selection criteria are employed to generate, augment, or reduce test suites. For example, in the research described later in this thesis, test cases for certain programs were generated based on the branch coverage adequacy criterion, meaning that our goal was to create test suites that exercised all reachable branches. In another case, for several large programs, a function-based test adequacy criterion was used, meaning that our goal was to create test suites that exercised all specification-based functions.

1.2.2 Regression Testing

Regression testing is testing performed after modifications have been made to a program, it involves re-testing a program in order to re-establish some level of confidence in that program's correctness. Regression testing can be used in both the development and maintenance phases of the software life-cycle. In the development phase, regression testing can be employed to validate the software each time modifications are applied to the code. In the maintenance phase, after the software has been corrected, adapted to a new environment, or enhanced to improve its

performance, regression testing is performed to ensure that all modifications have altered software to conform to new specifications, and that no new faults have been introduced by changes [63, 114].

There are many different regression-testing-related processes. One is the *batch process* where regression testing is the last part of the creation of a new release. In this process, after a new release of software is produced, regression testing is employed to remove faults and achieve necessary confidence in the software. A second process is the *incremental process* where regression testing is performed continuously, in cycles (e.g., nightly), in parallel with coding.

Leung and White [114] distinguish two types of regression testing: progressive regression testing and corrective regression testing. In progressive regression testing, specifications change to accommodate the need for new functionality, altered data formats, etc. Usually, new modules are created and added to the software. Progressive regression testing tests a modified program using modified specifications. Here, some test cases can become obsolete (no longer applicable) and some test cases must be added to test new functionality. In corrective regression testing, specifications do not change. Because specifications do not change, no tests become obsolete. However, because of code changes, an existing test suite may become insufficient according to the old adequacy criterion, so new tests are likely to be needed.

Leung and White [114] outline several important differences between development testing and regression testing.

- Previous test suites may be available for regression testing (of course, these may need to be augmented with additional test cases), while development testing requires developing the test suite.

- While development testing validates the whole program, regression testing needs to validate only parts of the program that are affected by modifications.

- Whereas development testing is typically allocated resources and time as part of software development budget, regression testing often has no budget or allocated time. Thus, engineers try to cut regression testing time as much as possible.

- In development testing, knowledge about software is readily available from the developers. In regression testing, this information is often not easily available, making the regression testing process more difficult and less efficient.

- The completion time for regression testing is usually less than that for development testing, because only a fraction of the program is verified.

- While development testing is only performed during the software development phase, regression testing is performed many times during the maintenance phase after the software is released.

Despite these differences between development testing and regression testing, there are also similarities between them. They both have the same overall purposes: to increase confidence in the software and find errors in the code.

1.2.3 Regression Test Selection and Test Suite Reduction

Engineers frequently reuse the test suite for different regression testing sessions. New features are tested by generating new test cases and adding them to the regression testing test suite. As the test suite grows, a simple retest-all approach (using the whole test suite in the regression testing) can become too expensive. As a result, to deal with the increasing cost of the regression testing, several methods have been proposed.

One such method is regression test selection (RTS) [115, 168]. In general, regression test selection uses program and test suite properties (such as trace information and change information) and selects a subset of the test suite for use in validation of the newly created software version. More formally, consider program P, modified version P', and the test suite T for P. Regression test selection selects test suite $T' \subseteq T$ which is used in regression testing of P' against P. A variety of RTS techniques have been developed so far and substantial research has been done in this area [9, 14, 18, 30, 55, 66, 67, 76, 113, 157, 158, 162, 163, 164, 165, 168, 169, 170, 171, 195, 219]. In [168], Rothermel and Harrold, give an overview of various RTS techniques.

Another method for reducing regression testing cost is test suite reduction [25, 56, 73, 147, 173, 212, 215, 216, 217, 218]. Like RTS, test suite reduction selects a part of the test suite to be used in future regression testing. Formally, reduction splits the original test suite T into two subsets: T_{old} and T_{new} where $T_{old} \cup T_{new} = T$ and $T_{old} \cap T_{new} = \emptyset$. In reduction, however, subset T_{old} is permanently discarded and never used again, while subset T_{new} is kept and used in future regression testing sessions. One of the advantages of reduction relative to selection is that it decreases

the amount of managed data (such as saved outputs and traces). In some instances, reduction and selection can be used together.

1.2.4 The Test Case Prioritization Problem

Regression test selection and test suite reduction reduce the cost of regression testing. Earlier studies, however, showed that reduced test suites could miss some faults which would otherwise be detected by the original test suite [173, 176, 212]. In other cases, regulations require engineers to execute all test cases (e.g., in safety critical systems). In these cases, omitting test cases may be risky, or not allowed.

If engineers must execute all test cases, which order of test cases should be used? One test order can be better than another under some metric. For example, a test suite under one order may reveal a given fault after executing ten test cases, while this test suite under a different order reveals the same fault after executing two test cases. Because a test suite under the second order reveals the fault earlier, it might give engineers more time to correct this fault. In some cases, if regression testing is terminated prematurely due to some unexpected event, a better test ordering would allow more valuable test cases to be executed before testing ends.

Test case prioritization orders a test suite to maximize some objective function defined on test orderings. We define the test case prioritization problem as follows:

> *The Test Case Prioritization Problem:*
>
> *Given*: T, a test suite; PT, the set of permutations of T; and 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 possible prioritizations (orders) of T, and f is an objective function that, applied to any such order, yields an *ordering quality* value for that order.

There are many possible goals for prioritization. For example, testers may wish to increase the coverage of code in the system under test at a faster rate, increase their confidence in the reliability of the system at a faster rate, or increase the rate at which test suites detect faults in that system during regression testing. In the definition of the test case prioritization problem, f represents a quantification of such a goal.

Rothermel et al. distinguish two varieties of test case prioritization: *general* and *version-specific* [178]. With general prioritization, given program P and test suite T, the test cases are prioritized in T with the aim of finding an order of test cases that will be useful over a succession of subsequent modified versions of P. With version-specific test case prioritization, given P and T, the test cases are prioritized in T with the aim of finding an order that will be useful on a specific version P' of P. In the former case, our hope is that the prioritized suite will be more successful than the original suite at meeting the goal of the prioritization *on average* over subsequent releases; in the latter case, our hope is that the prioritized suite will be more effective than the original suite at meeting the goal of the prioritization *for P' in particular.*

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 of test suites, test cases might be prioritized in terms of the extent to which they execute modules that have tended to fail in the past. Alternatively, test cases might be prioritized in terms of their increasing cost-per-coverage of code components, or

in terms of their increasing cost-per-coverage of features listed in a requirements 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.

Finally, although we focus in this thesis on prioritization for regression testing, prioritization can also be employed in the initial testing of software [4]. An important difference between these applications is that, in regression testing, prioritization techniques can use information gathered in previous runs of existing test cases to prioritize test cases for subsequent runs; such information is not available during initial testing.

1.2.5 Measurement of Prioritization 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 this purpose, Rothermel et al. defined a metric, APFD, which represents the weighted average of the percentage of faults detected during the execution of the test suite [34, 175]. APFD values range from 0 to 100; higher values imply faster (better) fault detection rates.

For purpose of illustration, 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 1.1.A. Suppose we place the test cases in order **A–B–C–D–E** to form a prioritized test suite $T1$. Figure 1.1.B shows the percentage of detected faults versus the fraction of the test suite $T1$ 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 test suite $T1$ has

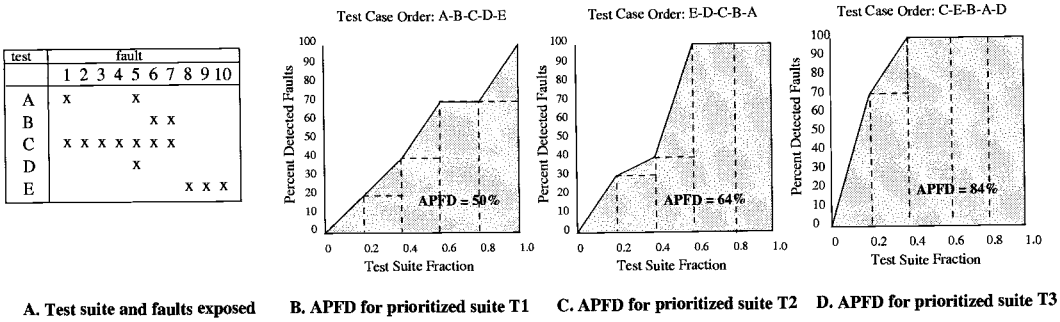


FIGURE 1.1: Examples illustrating the APFD metric.

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 the test suite has been used. 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 1.1.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 1.1.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%.

The formula for APFD for the test suite T under the given order is the following:

$$1 - \frac{TF_1 + TF_2 + ... + TF_m}{nm} + \frac{1}{2n} \tag{1.1}$$

where n is the number of tests, m is the number of faults, and TF_i is the test number (starting from one) which first reveals the fault i in the test suite T under the given order.

1.2.6 Previous Test Case Prioritization Techniques

Rothermel et al. [175, 178] describe several test case prioritization techniques. We next describe these techniques in detail; they include total statement coverage prioritization, additional statement coverage prioritization, total branch coverage prioritization, additional branch coverage prioritization, total fault-exposing-potential prioritization, and additional fault-exposing-potential prioritization.

The prioritization techniques that are described in this section, as well as these described later in this thesis, have been given different names in the literature. For consistency in this thesis, we impose a general naming scheme on all of these techniques. Details on this scheme and the names used are given in Appendix A. Because complete technique names are long, we also assign a mnemonic to each name.

1.2.6.1 Total Statement Coverage Prioritization

Total statement coverage prioritization (st-cov-nofb) orders test cases by decreasing number of covered statements. Test cases that cover the same number of statements are ordered randomly.

1.2.6.2 Additional Statement Coverage Prioritization

Additional statement coverage prioritization (st-cov-fb) keeps a vector of information telling whether a given statement has been covered. This technique marks all statements in the vector as *uncovered* and performs the following sequence of steps until no more unselected test cases remain: (1) if no test case can add any more statement coverage, mark all statements in the vector "*uncovered*"; (2) select an un-

selected test case that covers the largest number of statements uncovered so far (if several test cases cover the same number of uncovered statements, pick the next test case randomly); (3) update the vector, marking statements covered by the selected test case "*covered*"; (4) append the selected test case to the ordering.

1.2.6.3 Total Branch Coverage Prioritization

Total branch coverage prioritization (br-cov-nofb) operates like total statement coverage prioritization, but orders test cases by decreasing number of covered branches.

1.2.6.4 Additional Branch Coverage Prioritization

Additional branch coverage prioritization (br-cov-fb) is similar to additional statement coverage prioritization, except that it operates on branches instead of statements.

1.2.6.5 Total Fault-exposing-potential Prioritization

During testing, faults in certain locations are easier to detect than in others. Coverage alone is not always a sufficient metric by which to infer whether a fault in a given location is detected by a given test case. Execution of a faulty statement by a given test case is a necessary but not sufficient condition for detection of a fault hiding in this statement. PIE analysis, suggested by Voas [193], lists three necessary and sufficient conditions for a fault to be revealed in a given location. A faulty location must be executed, a data state just after the executed faulty location must be erroneous, and this erroneous data state must propagate to output. Three

probabilities, execution, infection, and propagation, correspond to these three conditions. Voas suggests a method for estimating these three probabilities, and defines a combined probability, called the *sensitivity* of a location, defining the probability for a location to reveal a fault if one exists, when executing a test case from a given profile.

More generally, the probability that a given test case reveals a fault in a given location, if one exists, is called its fault-exposing-potential (FEP).

One way to compute an approximation of FEP is to employ Voas's method. However, Voas's method for computing sensitivity is too conservative, and it is very difficult in practice to compute infection and propagation probability.

Alternatively, an approximation of each statement's FEP can be obtained by applying mutation analysis [30, 72]. In mutation analysis, for every location l and every test case t, the fault-exposing-potential approximation, $FEP_{l,t}$, is computed. $FEP_{l,t}$ is computed by the following set of steps: (1) produce set M_l of semantically different mutants in location l, (2) for every $m \in M_l$, compute a mutated program P^m and test P^m with test case t against the original program P, (3) compute $FEP_{l,t}$ as ratio of mutants *killed* (detected) by test case t over the total number of mutants $|M_l|$ (if t did not execute l, $FEP_{l,t} = 0$).

Total fault-exposing-potential prioritization (st-fep-nofb) works as follows. For every test case t, it computes $a_t = \sum_{l \in L} FEP_{l,t}$, where L is the set of all statements. Then, it sorts test cases in order of decreasing a_t.

Due to the high cost of mutation analysis used to compute fault-exposing-potential for every location and every test, fault-exposing-potential-based techniques are expensive. However, if the technique proves useful, less expensive approximations might be sought.

1.2.6.6 Additional Fault-exposing-potential Prioritization

Additional fault-exposing-potential prioritization (st-fep-fb) employs an iterative approach, similar to additional statement coverage prioritization. At each iteration, a test case t_{selected} is selected and the award values for other test cases are lowered due to increased confidence in the correctness of the locations covered by the selected test case. This is because, once a test case t is executed and succeeds, the probability that an executed statement does not contain a fault increases. A "confidence" vector is used to hold information for every statement, reflecting current confidence in its correctness. More precisely, the technique sets the confidence vector to zeroes and performs the following sequence of steps until no more unselected tests are left: (1) compute an award value for every test case using the formula $award_t = \sum_l ((1 - confidence_l) \times FEP_{l,t})$; (2) select the test case with the highest award value and add it to the ordered sequence; (3) for every statement l covered by selected test case t_{selected}, update $confidence_l = 1 - (1 - confidence_l) \times (1 - FEP_{l,t_{\text{selected}}})$.

1.2.7 Early Empirical Studies

In [175], Rothermel et al. describe experiments with several C programs ranging in size from 138 to 6218 lines of code.[2] Six prioritization techniques were studied including total statement coverage prioritization, additional statement coverage prioritization, total branch coverage prioritization, additional branch coverage prior-

---

[2] These were the Siemens and Space programs, described in detail in Section 2.4.2.

itization, total fault-exposing-potential prioritization, and additional fault-exposing-potential prioritization. All these techniques showed some ability to improve APFD relative to random orderings. Anova and Bonferroni analyses were performed to see whether statistical significance could be observed among different techniques. As overall results, for all subjects and versions combined, fault-exposing-potential techniques provided the best improvement in the rate of fault detection measured by the APFD metric, followed by the total branch and total statement coverage techniques.

In [178], Rothermel et al. extend these early results. The same programs used in [175] were used to perform several additional controlled studies. Because the faults studied in [175] provided a limited subset of the possible faults which could occur in practice, these additional studies also considered a substantial number of mutants (altered statements) as faults. In this study, prioritization techniques also showed improvement in the rate of fault detection relative to random test case orderings. Similar to the earlier experiment, in this case, additional fault-exposing-potential prioritization achieved statistically significantly better results than other techniques.

1.2.7.1 Work by Other Researchers

Several other researchers have considered the test case prioritization problem; in most cases this work postdates much of the research reported in this thesis, but we summarize it here.

Wong et al. Wong et al. [213] suggest prioritizing test cases according to the criterion of "increasing cost per additional coverage". In other words, given test

suite T and cost c_t for every test case $t \in T$, their technique performs the following steps: (1) for every $t \in T$, compute award value[3] a_t as the number of additionally covered statements divided by test cost c_t, (2) select test case t_s with the highest award value, (3) remove test case t_s from T, (4) add t_s to the ordered sequence, (5) keep track of statements uncovered so far, (6) go to step 1 if $T \neq \emptyset$. The authors restrict their attention to prioritization of the subset of test cases selected from a test suite by a safe regression test selection technique, and the selected test cases are only those test cases that reach modified code, but other test cases could be placed after these for later execution. No empirical results were reported on this prioritization technique.

Jones and Harrold. Jones and Harrold [96] describe a technique for prioritizing test cases for use with the modified condition/decision coverage (MCDC) test adequacy criterion. This technique uses an iterative approach, updating test information as test cases are added to the ordered sequence. Their algorithm generates a list of ordered sequences of test cases. Given test suite T and set of entities E, the algorithm works as follows: (1) mark all $e \in E$ uncovered; (2) $\forall t \in T$ compute $contribution_t$ (the sum of the MC/DC pairs completed and the number of entries, exits, and cases covered by test case t); (3) if $\forall t \in T$ $contribution_t = 0$, mark all $e \in E$ uncovered and select test case t_{selected} with the highest entity coverage; otherwise, select test case t_{selected} with the highest $contribution_{t_{\text{selected}}}$; (5) add test

[3] We use this term to define the test case worth metric, facilitating test case comparison where the test case with the highest award value is chosen as the next test case to be added to the ordered test suite.

case t_{selected} to the prioritized sequence; (6) mark all entities in E covered that are covered by t_{selected}; (7) remove t from T; (8) go to step 2 if $T \neq \emptyset$.

Srivastava and Thiagarajan. Srivastava and Thiagarajan [185] present a technique for prioritizing test cases based on basic block coverage. In their algorithm, during each iteration, a test case that covers the maximum number of uncovered yet changed basic blocks of code is selected. When no new changed blocks can be covered, the set of covered blocks is reset, and a new sequence is started. The technique differs from previous techniques in that it computes flow graphs and coverage from binary code, and attempts to predict possible affects on control flow following from code modifications. The authors describe the application of this technique to several large systems at Microsoft, and provide data showing that the approach can be applied efficiently to those systems. The authors also provide data suggesting that their prioritized test case orders achieve coverage quickly, and can detect faults early; however, their studies do not compare their prioritized test suites to other prioritization techniques and random ordering, so it is not possible to say whether the results represent an improvement in rate of fault detection over those that would be obtained under other orderings.

Kim and Porter. Kim and Porter [103] present a technique, which they refer to as a "history-based prioritization" technique, in which information from previous regression testing cycles is used to better inform the selection of a subset of an existing test suite for use on a modified version of a system. This technique is not, however, a "prioritization technique" in the sense defined in Section 1.2.4, because it imposes no ordering on test cases – the characteristic essential to the definition

of prioritization. Rather, the approach selects a subset of a test suite, using history information to determine which test cases should be selected, and is more accurately described as a "regression test selection technique" [168]. The information used can be test case execution, fault detection, or entity coverage (statement, branch, etc.). Given a set Pr of selection probabilities, a regression test selection technique X, and the original test suite T, their overall regression testing technique works as follows: (1) compute test suite $T_{\text{selected}} \subseteq T$ using X, (2) draw test case $t_{\text{selected}} \in T_{\text{selected}}$ that has the highest value $Pr_{t_{\text{selected}}}$, (3) execute t_{selected}, (4) go to step 2 if testing time is not over.

Avritzer and Weyuker. Avritzer and Weyuker [4] present techniques for generating test cases that apply to software that can be modeled by Markov chains, provided that operational profile data are available. Although the authors do not use the term "prioritization", their techniques generate test cases in an order that can cover a larger proportion of the software states most likely to be reached in the field earlier in testing, essentially, prioritizing the test cases in an order that increases the likelihood that faults more likely to be encountered in the field will be uncovered earlier in testing. Though not concerned with the prioritization of existing test cases for use in testing modified software, the approach provides an example of the application of prioritization in the case in which test suites are not available.

1.3 Open Problems

As Section 1.2.7.1 shows, in recent years more and more researchers have become interested in test case prioritization, developed new techniques, and explored their

effectiveness under various scenarios. However, prior to the start of this research, several open problems existed for prioritization. (These problems were described briefly at the end of Section 1.1. Here, we describe them in greater detail.)

First, only a limited set of techniques had been proposed. These include techniques based on statement and branch coverage and fault-exposing-potential. With the exception of Wong et al.'s technique, these techniques did not use information about program modifications, essentially prioritizing test suites without regard to a particular release. The problem with this approach is that while one ordering can be the best for one version, a different ordering may be better for a different version. If modifications are considered, this could allow techniques to target a particular software release, potentially producing orderings of better quality relative to a given software version.

Further, previous techniques have all used statement or branch-level coverage information. While this could be acceptable for small and medium sized programs, such a low granularity may not be practical for large software systems. Thus, techniques operating at higher granularity levels (e.g., performing analysis at the function level) may be used. Finally, for all such techniques, more extended studies need to be performed if results are to be generalized.

Previous research has also relied on an assumption that all test cases are equally expensive and all faults are equally severe. While this assumption can be appropriate under some circumstances, in many cases, variations in costs and severities cannot be ignored. Only Wong et al. in [213] informally mention a prioritization technique that uses test costs, but no studies were made to assess its effectiveness. Techniques that utilize estimations of test cost and fault severity need to be developed and studied.

Previously, prioritization techniques have been devised, described, and experimented with independently from each other, ignoring the fact that they have much in common. No work has been done to explore their commonalities and to devise a model or framework which could express those techniques. If a unifying framework for prioritization were developed, it could potentially help us implement a general algorithm whose instantiations produce a variety of prioritization techniques.

All previous research on prioritization has implicitly assumed that prioritization will be beneficial, if it increases the rate of fault detection. But this assumption is incorrect. Prioritization requires additional program preparation, data gathering, and application of prioritization tools. Also, a prioritized test suite, in some cases, may not be better than the original test case order. Thus, it is necessary to provide cost and benefit measures for prioritization techniques, to be able to decide whether it would be beneficial to apply prioritization at all, and if so, which technique would be the most economical.

Prioritization technique effectiveness varies greatly under different circumstances, such as across different programs, versions, faults, and types of modifications. There are a variety of factors underlying these circumstances that affect prioritization effectiveness. No work has been done to explore which factors affect prioritization effectiveness and how.

There are many different prioritization techniques to choose from. One technique can be the best in one case, while another technique can be better in other cases. The relative performance of prioritization techniques depends on a number of factors. No work has been done to facilitate selection of a technique that would achieve the best results.

In this work, our goal is to consider these open problems and resolve them.

1.4 Dissertation Overview

The remainder of this dissertation is organized as follows.

In Chapter 1.4, we present prioritization techniques that incorporate change information and work at different granularity levels (levels of analysis of code). We also present experiments performed to assess these techniques' effectiveness.

In Chapter 3, we present prioritization techniques and a new metric to assess orderings that take into account arbitrary test costs and fault severities. A case study is presented to assess the effectiveness of such techniques, explore different ways to estimate fault severities, consider several distributions of test costs and fault severities, and answer several practical questions about cost-cognizant prioritization.

In Chapter 4, we present a general prioritization technique framework and show how it describes all prioritization techniques published to date.

In Chapter 5, we present a prioritization cost model that can express the cost-benefit tradeoffs of prioritization. We show how to determine whether a given prioritization technique will be cost-effective or detrimental if applied under given circumstances. Finally, we present a case study of this cost model.

In Chapter 5.4, we consider two sets of factors that affect prioritization: test suite composition and code modifications. We describe experiments that study the impact of these factors on the effectiveness of several prioritization techniques. This lays the groundwork for developing practical guidelines to improve regression testability in terms of prioritization effectiveness.

In Chapter 7, we show how to apply decision trees to find a set of metrics defined on the program and its test suite that affect the relative performance of prioritization

techniques. This can provide an approach by which metrics can be used to predict in advance which technique will be the best to employ.

Finally, in Chapter 8, we conclude and describe opportunities for future work.

CHAPTER 2

INCORPORATING GRANULARITY AND MODIFICATION INFORMATION INTO PRIORITIZATION

2.1 Introduction

While statement coverage, branch coverage, and fep-based techniques produced favorable results, their application is rather limited. One issue is that these techniques do not take modifications into account when trying to produce a test ordering.[1] They do not use modification information, which may mean they are not as good at version specific prioritization as we might like.

A second issue is that all previously developed techniques operate at very low granularity where code analysis is concerned, such as statement and branch levels. While such low level information provides prioritization techniques with plentiful information about test case execution behavior, it may prove to be impractical, especially for large programs, due to several problems.

The first problem is that in order to obtain a statement or branch level trace (a list of statements or branches executed by a given test case), a program has to be significantly altered (instrumented) which could change its behavior, invalidating computed trace information; in other words, this method can be too invasive. This could happen for two reasons. The first reason is that low level instrumentation

---

[1] Portions of this chapter have appeared previously in [34, 48].

can affect timing and memory usage because extra statements are inserted into the source code and the resulting executables, at runtime, require additional memory. Different timing and memory usage can potentially affect program behavior due to synchronization problems and violations of programmers' assumptions about time and memory usage.

The second problem is that low granularity coverage information requires a lot of storage capability making it expensive for large programs.

The third problem is that the more coverage information we have, the more time is needed to perform prioritization. Prioritization time depends on the amount of coverage data, and large amounts of data can make prioritization unacceptable due to the large amount of time it demands.

Higher level instrumentation is not immune to these problems, but we expect it to be less affected.

As a result, to make fine granularity based techniques usable for large programs, coverage information may have to be collected and used at a higher level than statements or branches, such as the level of functions. For larger real-world software systems consisting of millions of lines of the code, even higher granularity coverage information might be collected, such as at the level of classes (objects), modules, files, or even individual programs composing the system.

In this section, we present test case prioritization algorithms that utilize change information and can work at arbitrary coverage granularity levels.

2.2 Change Metrics

Change metrics measure the locations and amount of change made to produce a given software version. Change information can be used in the prioritization process to give preference to test cases that exercise changes. Our goal is to give preference to tests which are more capable of revealing *regression faults* - faults occurring as the result of changes. Because regression faults occur solely in changed code, our hypothesis is that there is a positive correlation between the amount of modified code that a given test executes and the fault revealing capability of a test case. Our heuristic in change-based prioritization techniques is to order tests in accordance with the amount of modified code they may contact.

Because of the difficulty of defining and measuring change in individual statements or branches, we computed and used change information at the level of functions. (We could also use the level of modules, files, objects, or programs that comprise a large software system.)

To measure the amount of change in a given function, we employed four methods: binary textual code differencing (binary diff-based metric), diff-based textual code differencing (diff-based metric), fault index differencing (fi-based metric), and binary fault index differencing (binary fi-based metric).

2.2.1 Textual Code Differencing

The simplest type of change information we compute is binary data reflecting whether a change has occurred or not. In binary textual code differencing, we compare corresponding program components (functions, files, modules). If a given component has been modified in the new version, its binary change information is

assigned one, otherwise it is assigned zero. To make this measure more effective, we could also assess whether a given difference has any effect on the resulting compiled code. For example, changes can include beautification formatting such as adding or removing white space, modifying comments, merging or splitting lines, etc. On the other hand, we have to make sure that any change that can affect code generation is reflected by this metric, to be conservative.

The advantage of binary textual code differencing is its simplicity: it requires minimal implementation and is easy to collect. Because in many cases, programmers and configuration management systems keep change logs, indicating which components have been modified, it can be easy to obtain binary change information.

Such simplicity, however, comes at a cost. When programs are changed, the number of faults may differ from one changed location to another. Accepting the assumption that regression fault density is uniform across changed code, we can expect that the more changes a location has, the higher the expected number of faults it may contain. Binary differencing does not reflect this reality.

One way to address this problem is to collect non-binary textual difference information on every location, using a simple *diff* tool applied to the two versions of each location. The number of added, deleted, and modified lines of code could be one reasonable metric for the amount of difference.

This diff-based metric is also relatively inexpensive, requiring applying a diff tool to each pair of corresponding locations. However, it is more expensive than simple binary differencing because of the additional work required to collect the information and automate the process.

2.2.2 *Fault Index Differencing*

Textual differencing has three drawbacks. First, it measures the number of line-level modifications. While it is easy to comprehend this metric, the metric assumes that the probability of a fault's existence is linearly related to the amount of change. While we may reasonably expect to see a correlation between fault proneness and amount of change, this correlation may not be linear. Second, some types of changes are more fault prone than others: replacing identifier names to improve code readability is far less fault prone than optimizing an algorithm, despite the fact that in the first case, there can be a greater amount of change than in the second. Thus, we should take into account how modification type affects regression fault density. Third, faults are not equally likely to exist in each location; rather, certain locations are more likely to contain faults than others. In other words, the same changes may have different fault proneness at different locations. For example, changing one parameter's type in a function can be more error prone if this function has many similar parameters than if the function has just one parameter.

One way to address these drawbacks is to use fault proneness metrics associated with measurable software attributes [2, 7, 21, 101, 126]. In the context of regression testing, we are interested in the potential influence, on fault proneness, of our modifications; that is, with the potential of modifications to lead to regression faults. This requires that our fault proneness measure accounts for attributes of software change [39]. Given such a measure, we can account for the association of change with fault-proneness by prioritizing test cases based on this measure.

For this technique, as a metric of fault proneness, we use a *fault index* which, in previous studies [39, 145], has proven effective at providing fault proneness es-

timates. We compute this metric per function. The fault index generation process involves the following steps. First, a set of measurable attributes [38] is obtained from each function in the program. Second, the metrics are standardized using the corresponding metrics of a baseline version (which later facilitates the comparison across versions). Third, principal components analysis [95] reduces the set of standardized metrics to a smaller set of domain values, simplifying the dimensionality of the problem and removing the metrics collinearity. Finally, the domain values weighted by their variance are combined into a linear function to generate one fault index per function in the program.

Given program P and subsequent version P', generating regression fault indices for P' requires generation of a fault index for each function in P, generation of a fault index for each function in P', and a function-by-function comparison of the indexes for P' against those calculated for P. As a result of this process, the regression fault proneness of each function in P' is represented by a regression fault index based on the complexity of the changes that were introduced into that function. Further details on the mechanisms of the method are given in [39, 61]. From this point onward and to simplify the nomenclature, we refer to "regression fault indexes" simply as "fault indexes".

Fault index computation requires more complicated program analysis and requires additional tool development beyond textual difference. If it can be shown to be effective, however, tools for efficiently computing fault indexes could be developed.

Similar to binary textual differencing, a binary fault index can be developed, distinguishing only between zero and nonzero fault indices. We call this a "binary fault-index-based metric".

technique	locations	change	feedback	update-_conf	award_term-_func
st-cov-nofb	statement	unit	false	false	FUNC1
st-cov-fb	statement	unit	true	false	FUNC1
fn-cov-nofb	function	unit	false	false	FUNC1
fn-cov-fb	function	unit	true	false	FUNC1
st-fep-nofb	statement	unit	false	false	FUNC2
st-fep-fb	statement	unit	true	true	FUNC2
fn-fep-nofb	function	unit	false	false	FUNC2
fn-fep-fb	function	unit	true	true	FUNC2
fn-mod-cov-nofb	function	change	false	false	FUNC1
fn-mod-cov-fb	function	change	true	false	FUNC1
fn-mod-fepm-nofb	function	change	false	false	FUNC3
fn-mod-fepm-fb	function	change	true	false	FUNC3
fn-mod-fep-nofb	function	change	false	false	FUNC4
fn-mod-fep-fb	function	change	true	false	FUNC4

TABLE 2.1: Prioritization Techniques and Parameters for use with Algorithm 1.

2.3 Prioritization Techniques

We wished to incorporate granularity and modification information into prioritization. Rather than create many different algorithms, we created a single algorithm, Algorithm 1, that expresses the full range of techniques that we wish to support. This is achieved by invoking the algorithm with the proper parameters.

Table 2.1 lists the various techniques expressed by Algorithm 1, and the parameter values required to invoke each technique. The final parameter, $award\_term\_compute$, supplies a function used in award value computation for computing a single term. This function can be any of the functions: $FUNC1$,

Algorithm 1 Test case prioritization.

Procedure **Prioritize**
In: $coverage$ (location-wise binary coverage),
$locations$ (set of program locations),
$tests$ (set of test cases),
fep (fault-exposing-potential),
$_{orig}change$ (change information),
$feedback$ (parameter specifying whether feedback is to be used),
$update\_conf$ (parameter specifying whether to update confidence vector),
$award\_term\_func$ (parameter function which computes a single term for award value)
Out: $list$ (prioritized list of test cases)

$list = \epsilon$
for all $l \in locations$ **do**
  $change_l = {}_{orig}change_l$
  $confidence_l = 0$
end for
while $tests \neq \emptyset$ **do**
  $awards$ = **Compute_Award_Values**($change, fep, confidence, coverage, tests, locations, award\_term\_func$)
  if all award values are zeroes **then**
    $change$ = **Reset**($_{orig}change, locations$)
    $awards$ = **Compute_Award_Values**($change, fep, confidence, coverage, tests, locations, award\_term\_func$)
  end if
  find the test $t_{\text{selected}} \in tests$ with the largest award value $awards_{t_{\text{selected}}}$
  add t_{selected} to $list$
  remove t_{selected} from $tests$
  if $feedback = true$ **then**
    for all $l \in locations$ **do**
      if $coverage_{l,t_{\text{selected}}} = 1$ **then**
        $change_l = 0$
        if $update\_conf = true$ **then**
          $confidence_l = 1 - (1 - confidence_l) \times (1 - fep_{l,t_{\text{selected}}})$
        end if
      end if
    end for
  end if
end while
return $list$

Procedure **Compute_Award_Values**
In: $fep, change, confidence, coverage, tests, locations, award\_term\_func$
Out: $awards$

for all $t \in tests$ **do**
  $awards_t = 0$
  for all $l \in locations$ **do**
    $awards_t = awards_t +$ award_term_func($coverage_{l,t}, change_l, fep_{l,t}, confidence_l$);
  end for
end for
return $awards$

Procedure **Reset**
In: $_{\text{orig}}change, locations$
Out: $change$

for all $l \in locations$ **do**
  $change_l = {}_{\text{orig}}change_l$
end for
return $change$

$FUNC2$, $FUNC3$, and $FUNC4$, given in Figure 2.1. Function $FUNC4$ returns a tuple. For summation inside **compute_award_value**, we define addition of n tuples $< x_1, y_1 >, < x_2, y_2 >, ..., < x_n, y_n >$ to be $< \sum_{i=1}^{n} x_i, \sum_{i=1}^{n} y_i >$. To find the largest award value, if k-tuples are used, we select the tuple with the largest first element; if there is a tie, we select the tuple with the largest second element; and so on, until reaching the k-th element. If there is still a tie, we resolve it by selecting the tuple with the smallest test number.

In Table 2.1, the second column specifies whether the modification information is utilized by a given prioritization technique. If no such information is utilized (in cases of simple fep- or coverage-based techniques), unit values (each value is "1") are substituted instead (specified by "unit"). "Change" specifies the cases in which diff, binary diff, fault index, or binary fault index information is used.

For brevity, we list each change-based technique only once in the table. Thus, each change-based technique in the table corresponds to four different change-based techniques: FI, binary FI, diff, and binary diff. Thus, "mod" can be "fi", "bfi", "diff", or "bdiff."

2.4 Controlled Experiments

To illustrate and experimentally evaluate the effectiveness of prioritization techniques, we conducted a set of controlled experiments and a case study. Our controlled experiments use several small programs and one medium size program, and our case study uses two medium size programs and one large size program.

We present our controlled experiments first.

Function **FUNC1**
In: $coverage\_term$ (coverage data for a given location and test),
$change\_term$ (current change data for a given location)
fep_term (fep data for a given location and test)
$confidence\_term$ (current confidence data for a given location)
Out: $term$

$term = coverage\_term \times change\_term$
return $term$;

Function **FUNC2**
In: $coverage\_term$ (coverage data for a given location and test),
$change\_term$ (current change data for a given location)
fep_term (fep data for a given location and test)
$confidence\_term$ (current confidence data for a given location)
Out: $term$

$term = (1 - confidence\_term) \times fep\_term$
return $term$;

Function **FUNC3**
In: $coverage\_term$ (coverage data for a given location and test),
$change\_term$ (current change data for a given location)
fep_term (fep data for a given location and test)
$confidence\_term$ (current confidence data for a given location)
Out: $term$

$term = coverage\_term \times change\_term \times (1 - confidence\_term) \times fep\_term$
return $term$;

Function **FUNC4**
In: $coverage\_term$ (coverage data for a given location and test),
$change\_term$ (current change data for a given location)
fep_term (fep data for a given location and test)
$confidence\_term$ (current confidence data for a given location)
Out: $term$

$term1 = coverage\_term \times change\_term$
$term2 = (1 - confidence\_term) \times fep\_term$
return $< term1, term2 >$;

FIGURE 2.1: Specific award value term computation functions, used as parameters in Algorithm 1.

2.4.1 *Research Questions*

Our experiments address the following specific research questions.

RQ1: Can version-specific test case prioritization techniques improve the rate of fault detection of test suites?

RQ2: How do fine granularity (statement level) prioritization techniques compare to coarse granularity (function level) techniques in terms of the rate of fault detection?

RQ3: Can the use of predictors of fault proneness improve the rate of fault detection of prioritization techniques?

2.4.2 *Programs*

We used eight C programs, with faulty versions and a variety of test cases, as objects of study. Seven of these programs were assembled by researchers at Siemens Corporate Research for experiments with control-flow and data-flow test adequacy criteria [93]; we refer to these as the *Siemens programs*. The eighth program, `space`, was developed for the European Space Agency; we refer to this program as the *space program*.

Table 2.2 provides metrics on the programs; we explain the meaning of these metrics in the following paragraphs.

Program	Lines of Code	1st-order Versions	Test Pool Size	Test Suite Avg. Size
tcas	138	41	1608	6
schedule2	297	10	2710	8
schedule	299	9	2650	8
tot_info	346	23	1052	7
print_tokens	402	7	4130	16
print_tokens2	483	10	4115	12
replace	516	32	5542	19
space	6218	35	13585	155

TABLE 2.2: Experiment Objects

2.4.2.1 Siemens Programs

The Siemens programs perform various tasks: `tcas` models an aircraft collision avoidance algorithm, `schedule2` and `schedule` are priority schedulers, `tot_info` computes statistics, `print_tokens` and `print_tokens2` are lexical analyzers, and `replace` performs pattern matching and substitution. For each program, the Siemens researchers created a *test pool* of black-box test cases using the category partition method [8, 149]. They then augmented this test pool with manually created white-box test cases to ensure that each exercisable statement, edge, and definition-use pair in the base program or its control flow graph was exercised by at least 30 test cases. The researchers also created faulty versions of each program by modifying code in the base version; in most cases they modified a single line of code, and in a few cases they modified between two and five lines of code. Their goal was to introduce faults that were as realistic as possible, based on their experience with real programs. To obtain meaningful results, the researchers

retained only faults that were detectable by at least three and at most 350 test cases in the test pool.

2.4.2.2 *Space Program*

The `space` program is an interpreter for an array definition language (ADL). The program reads a file of ADL statements, and checks the contents of the file for adherence to the ADL grammar and specific consistency rules. If the ADL file is correct, `space` outputs an array data file containing a list of array elements, positions, and excitations; otherwise the program outputs error messages. The `space` program has 35 versions, each containing a single fault: 30 of these were discovered during the program's development, five more were discovered subsequently [178]. The test pool for `space` was constructed in two phases. The pool was initialized to 10,000 test cases randomly generated by Vokolos and Frankl [196]. Then new test cases were added until each executable edge in the program's control flow graph was exercised by at least 30 test cases. This process produced a test pool of 13,585 test cases.

2.4.2.3 *Test Suites*

Sample test suites for these programs were constructed using the test pools for the base programs and test-coverage information about the test cases in those pools. More precisely, to generate a test suite T for base program P from test pool T_p, the C pseudo-random-number generator `rand`, seeded initially with the output of the C `times` system call, was used to obtain integers that were treated as indexes into T_p (modulo $|T_p|$). These indexes were used to select test cases from T_p; each test case t

was added to T only if t added to the cumulative branch coverage of P achieved by the test cases added to T thus far. Test cases were added to T until T contained at least one test case that would exercise each executable branch in the base program. Table 2.2 lists the average sizes of the 1000 branch-coverage-adequate test suites generated by this procedure for each of the object programs.

For our experimentation, we randomly selected 50 of these test suites for each program.

2.4.2.4 Versions

For these experiments we required program versions with varying numbers of faults; we generated these versions in the following way. Each program was initially provided with a correct base version and a *fault base* of versions containing exactly one fault. We call these 1st-order versions. We identified, among these 1st-order versions, all versions that do not interfere – that is, all faults that can be merged into the base program and exist simultaneously. For example, if fault f1 is caused by changing a single line and fault f2 is caused by deleting the same line, then these modifications interfere with each other.

We then created higher-order versions by combining non-interfering 1st-order versions. To limit the threats to our experiment's validity, we generated the same number of versions for each of the programs. For each program, we created 29 versions; each version's order varied randomly between 1 and the total number of non-interfering 1st-order versions available for that program.[2] At the end of this

---

[2] The number of versions, 29, constitutes the minimum among the maximum number of versions

process, each program was associated with 29 multi-fault versions, each containing a random number of faults.

2.4.3 Prioritization Techniques

In this study, we used the following prioritization techniques:

Full Technique's Name	Mnemonic
statement coverage no feedback	st-cov-nofb
statement coverage feedback	st-cov-fb
function coverage no feedback	fn-cov-nofb
function coverage feedback	fn-cov-fb
function fault index coverage no feedback	fn-fi-cov-nofb
function fault index coverage feedback	fn-fi-cov-fb
function diff coverage no feedback	fn-diff-cov-nofb
function diff coverage feedback	fn-diff-cov-fb
statement fault-exposing-potential no feedback	st-fep-nofb
statement fault-exposing-potential feedback	st-fep-fb
function fault-exposing-potential no feedback	fn-fep-nofb
function fault-exposing-potential feedback	fn-fep-fb
function fault index fault-exposing-potential no feedback	fn-fi-fep-nofb
function fault index fault-exposing-potential feedback	fn-fi-fep-fb
function diff fault-exposing-potential no feedback	fn-diff-fep-nofb
function diff fault-exposing-potential feedback	fn-diff-fep-fb
random	random
optimal	optimal

The random technique orders test cases in a random order; it is used as a control technique. The optimal technique heuristically orders test cases to maximize the

---

that could be generated for each program given the interference constraints.

rate of fault detection using fault information; it is also used as a control technique.[3] Other techniques are described in Section 2.3.

2.4.4 Experiment Design, Results and Analysis

We performed several experiments, each addressing one of our research questions. Each experiment included five stages: (1) stating a research question in terms of a hypothesis, (2) formalizing the experiment through a robust design, (3) collecting data, and (4) analyzing data to test the hypothesis. In general, each experiment examined the results of applying certain test case prioritization techniques to each program and its set of versions and test suites.

To provide an overview of all the collected data[4] we include Figures 2.2 and 2.3 with box plots.[5] Figure 2.2 displays a plot for an "all programs" total, and Figure 2.3 displays an individual plot for each of the programs. Each plot contains a box showing the distribution of APFD scores for each of the 18 techniques.

The following sections describe, for each of our research questions in turn, the experiment(s) relevant to that question, presenting their design and the analysis of their results.

---

[3] An optimal ordering can only be computed after testing is completed and faults are known. Thus, it cannot be used as a prioritization technique, it is merely used as an upper bound on prioritization effectiveness. Because the problem of finding an optimal order is NP-hard, we used a heuristic. Thus, our estimate of the upper bound was conservative.

[4] For simplicity, data belonging to separate experiments are presented together.

[5] Box plots provide a concise display of a data distribution. The small rectangle embedded in each box marks the mean value. The edges of the box are bounded by the standard error. The whiskers extend to one standard deviation.

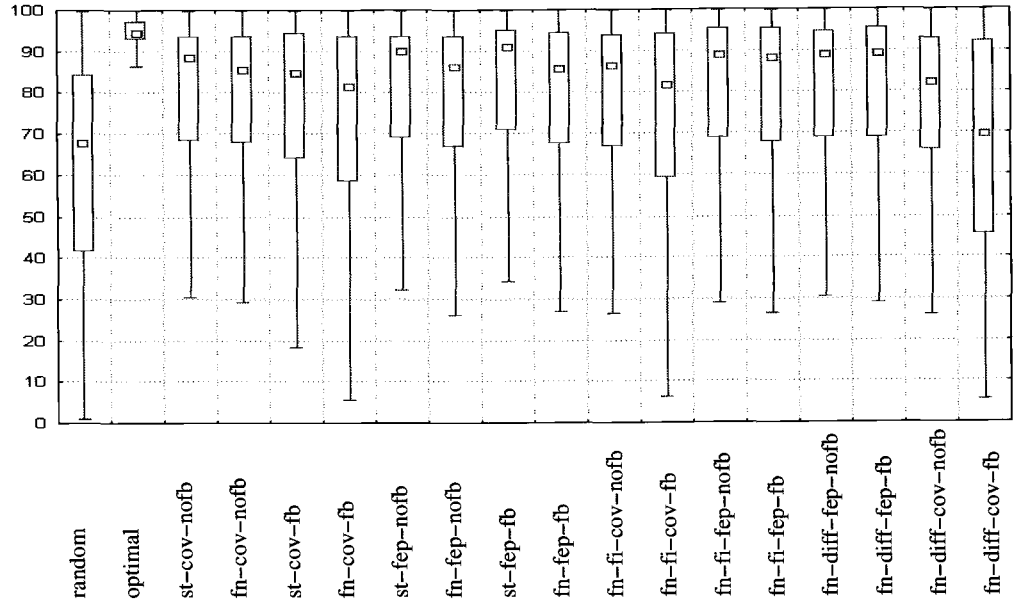


FIGURE 2.2: APFD boxplots for an "all programs" total. The horizontal axis lists techniques, and the vertical axes list APFD scores.

2.4.4.1 Experiment 1 (RQ1): Version-specific Prioritization

Our first research question considers whether version-specific test case prioritization can improve the fault-detection abilities of test suites. Since we conjectured that differences in the granularity at which prioritization is performed would cause significant differences in APFD values, we performed two experiments: Experiment 1a involving statement level techniques st-cov-nofb, st-cov-fb, st-fep-nofb and st-fep-fb, and Experiment 1b involving function level techniques fn-cov-nofb, fn-cov-fb, fn-fep-nofb and fn-fep-fb. This separation into two experiments gave us more power to determine differences among the techniques within each group.

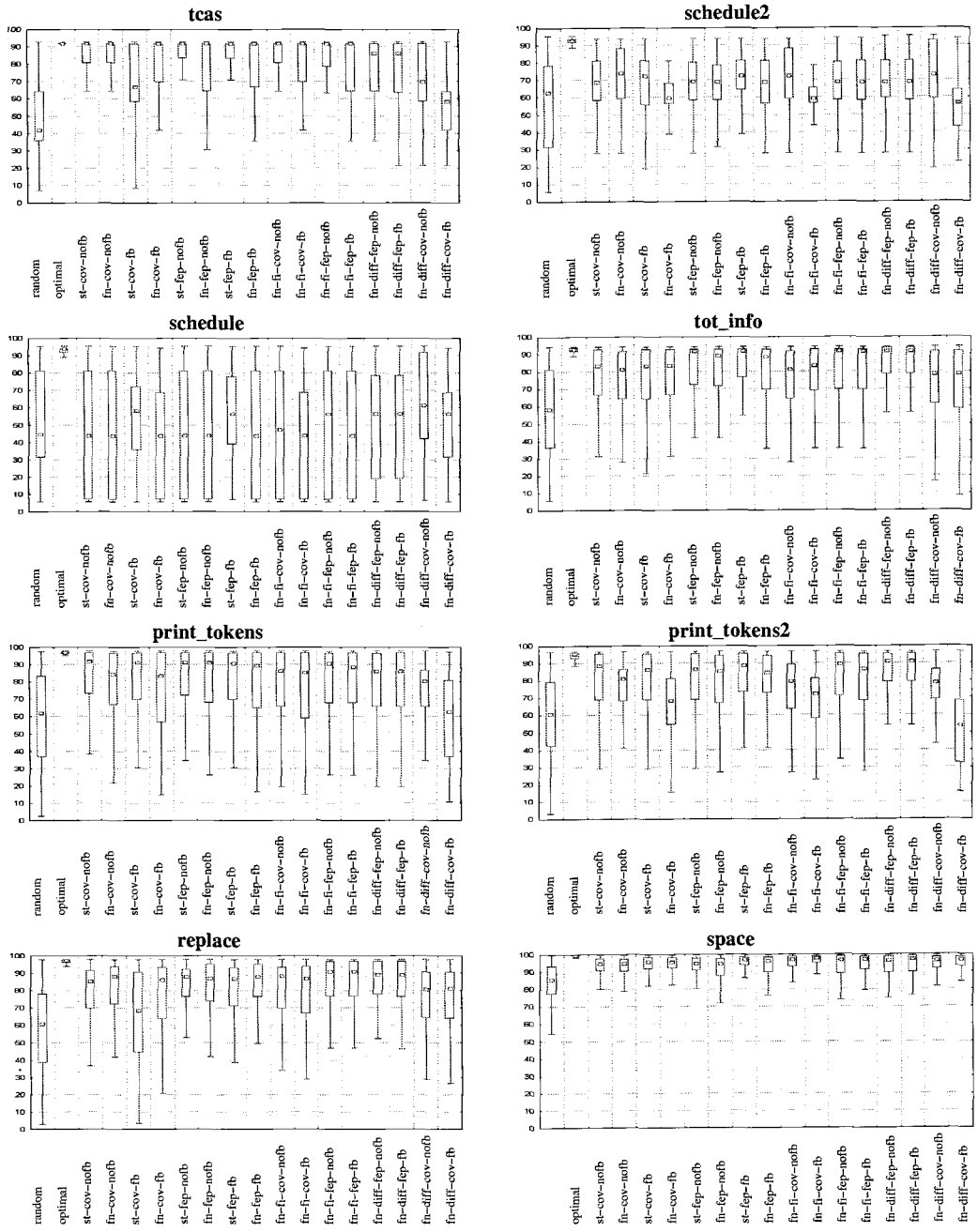


FIGURE 2.3: APFD boxplots for individual programs. The horizontal axes list techniques, and the vertical axes list APFD scores.

	SS	Degrees of freedom	MS	F	p
PROGRAM	3473054	7	496150.60	1358.512	0.00
TECHN	97408	3	32469.20	88.904	0.00
PROGRAM*TECHN	182322	21	8682.00	23.772	0.00
Error	9490507	25986	365.22		

TABLE 2.3: ANOVA Analysis, Statement Level Techniques, All Programs

Both experiments followed the same factorial design: all combinations of all levels of all factors were investigated. The main factors were program and prioritization technique. Within programs, there were 8 levels (one per program) with 29 versions and 50 test suites per program. We employed 4 prioritization techniques per experiment. Each treatment (prioritization technique) was applied to every viable combination of test suite and version within each program generating a maximum of 46400 observations (each including an APFD value) per experiment.

We then performed an analysis of variance (ANOVA) on those observations to test the differences between the techniques' mean APFD values. We considered the main effects program and technique, and the interaction among those effects. When the ANOVA F-test showed that the techniques were significantly different, we proceeded to determine which techniques contributed the most to that difference and how the techniques differed from each other through a Bonferroni multiple comparison method. This procedure works within the ANOVA setting to compare the techniques' means while controlling the family-wise type of error.

Experiment 1a: Statement Level. Table 2.3 presents ANOVA results for Experiment 1a, considering all programs. The treatments are in the first column, and the

sum of squares, degrees of freedom, and mean squares for each treatment are in the following columns. The F values constitute the ratio between the treatment and the error effect (last row). The larger the F statistic, the greater the probability of rejecting the hypothesis that the techniques' mean APFD values are equal. The last column presents the p-values, which represent "the probability of obtaining a value of the test statistic that is equal to or more extreme than the one observed" [105]. Since we selected our level of significance to be 0.05%, we reject the hypotheses when the p-value is less than or equal to that level of significance. Otherwise, we do not reject the hypothesis.

The results indicate that there is enough statistical evidence to reject the null hypothesis H1; that is, the means for the APFD values generated by different statement level techniques were different. However, the analysis also indicates that there is significant interaction between techniques and programs: the difference in response between techniques is not the same for all programs. Thus, individual interpretation is necessary. As a first step in this interpretation we performed an ANOVA on each of the programs. Each of the ANOVAs was significant, indicating that, within each program, the statement level prioritization techniques were significantly different. Table B.1 in Appendix A presents the results of these ANOVAs.

The ANOVA analysis evaluated whether the techniques differed, the APFD means ranked the techniques, a multiple comparison procedure using Bonferroni analysis quantifies how the techniques differed from each other. Table 2.4 presents the results of this analysis for all of the programs, ranking the techniques by mean. Grouping letters indicate differences: techniques with the same grouping letter were not significantly different. For example, st-fep-nofb has a larger mean than st-cov-nofb but they are grouped together because they were not significantly different. On

Grouping	Means	Techniques
A	80.733	st-fep-fb
B	78.867	st-fep-nofb
B	78.178	st-cov-nofb
C	76.077	st-cov-fb

TABLE 2.4: Bonferroni Means Separation Tests, Statement Level Techniques, All Programs

the other hand, the st-fep-fb technique, which uses FEP information and coverage with feedback, was significantly better than the other techniques. The last technique ranked is st-cov-fb, which was significantly weaker than the others.

To consider results on a per-program basis, we performed a Bonferroni analysis on each of the programs. Table B.2 in Appendix A presents the results of these analyses. On `replace`, st-fep-nofb, st-fep-fb, and st-cov-nofb ranked at the top but were not significantly different from each other. The same scenario held for `schedule2` and `tcas`. On `schedule`, the techniques that use feedback (st-fep-fb and st-cov-fb) ranked at the top but were not significantly different, while the techniques that do not use feedback (st-cov-nofb and st-fep-nofb) were significantly inferior. On `space`, st-fep-fb was significantly better than other techniques, while the rest of the techniques did not differ from each other. `Print_tokens` presented a unique case because the Bonferroni process could not find differences among any pair of techniques, even when the ANOVA specified that there was significant difference when the four of them were considered. On `print_tokens2`, st-fep-fb ranked at the top, followed by the other techniques among which there was no significant difference. Finally, `tot_info`'s ranking matched the overall ranking for

	SS	Degrees of freedom	MS	F	p
PROGRAM	4139625	7	591375.00	1501.327	0.00
TECHN	60953	3	20317.80	51.581	0.00
PROGRAM*TECHN	158227	21	7534.60	19.128	0.00
Error	10071668	25569	393.90		

TABLE 2.5: ANOVA Analysis, Basic Function Level Techniques, All Programs

all applications, although no significant difference was found between techniques using and not using feedback.

To summarize, although the rankings of techniques did vary somewhat among programs, similarities did occur across all or across a large percentage of the programs. Specifically, st-fep-fb ranked in the highest Bonferroni group of techniques independent of the program; st-fep-nofb and st-cov-nofb were in the same group (not significantly different) on seven of the eight programs; and finally, st-cov-fb ranked significantly worse than all other techniques on four programs.

Experiment 1b: Function Level. Table 2.5 presents the analysis of variance results for Experiment 1b (function level techniques) considering all programs. The interaction effects between techniques and programs were also significant for function-level techniques, and the results revealed significant differences among the techniques. Moreover, the techniques ranked in the same order as their statement-level equivalents, with fn-fep-fb first, fn-fep-nofb second, fn-cov-nofb third, and fn-cov-fb last. However, as shown by the results of Bonferroni analysis (Table 2.6), the top three techniques were not significantly different from each other.

Grouping	Means	Techniques
A	77.453	fn-fep-fb
A	76.957	fn-fep-nofb
A	76.928	fn-cov-nofb
B	73.465	fn-cov-fb

TABLE 2.6: Bonferroni Means Separation Tests, Basic Function Level Techniques, All Programs

Following the same steps as in Experiment 1a, we next performed ANOVA and Bonferroni analyses on a per program basis. Tables B.3 and B.4, in Appendix A, present the results of these analyses. The results on `replace`, `schedule`, `print_tokens`, and `tot_info` present trends similar to those seen in the Bonferroni results for all programs. On `print_tokens2`, the ranking was identical but all the techniques produced significantly different averages. `Schedule2`, `tcas` and `space` present a different perspective. On `schedule2` and `tcas`, fn-cov-nofb was significantly better than the other techniques. On `space`, fn-cov-fb was the best, fn-cov-nofb came second, and the FEP-based techniques followed.

In summary, for the function-level techniques, we observed great variation in the techniques' performance across subjects. The most surprising result was the lack of significant gains observed, for function-level techniques, when using FEP estimates. At a minimum, this suggests that our method for estimating FEP values at the function level may not be as powerful as our method for estimating those values at the statement level. Furthermore, at the function level, except for `print_tokens2`, the two FEP techniques were not significantly different from one another. This implies that feedback had no effect when employing function level FEP techniques.

We also observed that using feedback could have a negative impact on APFD values. There is a possible explanation for this. Techniques at the function level employing feedback give higher priority to test cases that execute uncovered functions, discarding functions already executed independently of the section or percentage of code in those functions that has actually been covered. If those partially covered functions are faulty but their faulty sections have not yet been covered, and the test cases executing those functions are given low priority by techniques with feedback, then APFD values for techniques employing feedback could be lower.

2.4.4.2 Experiment 2 (RQ2): Granularity Effects

Our second research question concerns the relationship between fine and coarse granularity prioritization techniques. Initial observations on the data led us to hypothesize that granularity has an effect on APFD values. This is suggested by comparing Table 2.4 to Table 2.6: for all cases the mean APFD values for function level techniques were smaller than the mean APFD values for corresponding statement level techniques (for example, the mean APFD for fn-fep-fb was 77.45, but for st-fep-fb it was 80.73). The radar chart in Figure 2.4 further illustrates this observation. In the radar chart, each technique has its own APFD value axis radiating from the center point. There are two polygons, representing the granularities at the statement and function levels, respectively. The radar chart shows that each function level technique had a smaller APFD than its counterpart at the statement level, and that statement level techniques as a whole were better (cover a larger surface) than function level techniques. The chart also shows that techniques employing feedback were more sensitive to the shift in granularity.

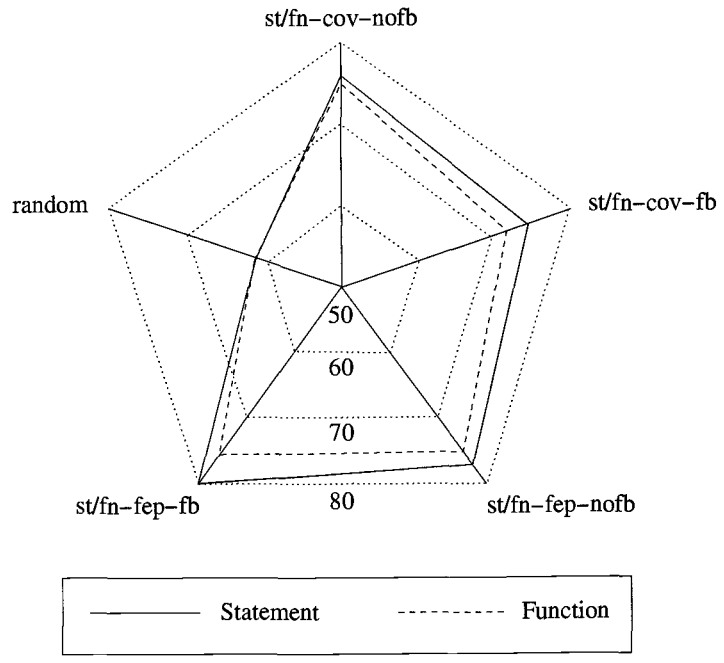


FIGURE 2.4: Radar chart (the line connects points whose distances from the center specify $APFD_C$ values).

To formally address this research question we performed a pairwise analysis among the following pairs of techniques: (st-cov-nofb, fn-cov-nofb), (st-cov-fb, fn-cov-fb), (st-fep-nofb, fn-fep-nofb), and (st-fep-fb, fn-fep-fb). The four orthogonal contrasts were significantly different as shown in Tables 2.7 and 2.8. That is, for these four pairs of techniques, different levels of granularity had a major effect on the value of the fault detection rate. Thus, in spite of the different rankings obtained in Experiments 1a and 1b, there is enough statistical evidence to confirm that statement level techniques were more effective than function level techniques.

Analyses on a per-program basis, shown in Tables B.5 and B.6, in Appendix A, present a similar picture. Although in several cases, statement-level techniques are

	SS	Degrees of freedom	MS	F	p
PROGRAM	7516093	7	1073728.	2829.748	0.00
TECHN	198981	7	28426.00	74.915	0.00
PROGRAM*TECHN	434621	49	8870.00	23.376	0.00
Error	19562175	51555	379.44		

TABLE 2.7: ANOVA Analysis, Function versus Statement Level Techniques, All Programs

Grouping				Means	Techniques
A				80.733	st-fep-fb
B				78.867	st-fep-nofb
B	C			78.178	st-cov-nofb
	C	D		77.453	fn-fep-fb
		D	E	76.957	fn-fep-nofb
		D	E	76.928	fn-cov-nofb
			E	76.077	st-cov-fb
F				73.465	fn-cov-fb

TABLE 2.8: Bonferroni Analysis, Function versus Statement Level Techniques, All Programs

not significantly better than their corresponding techniques (e.g., on `schedule`, st-cov-nofb and fn-cov-nofb do not differ significantly) only two cases occur in which a function-level technique significantly outperforms its corresponding statement-level technique. (These cases all involve st-cov-fb versus fn-cov-fb, and occur on `tcas` and `space`.)

	SS	Degrees of freedom	MS	F	p
PROGRAM	11860200	7	1694314.00	4580.131	0.00
TECHN	800070	11	72734.00	196.616	0.00
PROGRAM*TECHN	1220361	77	15849.00	42.843	0.00
Error	28551710	77182	369.93		

TABLE 2.9: ANOVA Analysis, All Function Level Techniques, All Programs

2.4.4.3 Experiment 3 (RQ3): Adding Prediction of Fault Proneness

Our third research question considered whether predictors of fault proneness can be used to improve the rate of fault-detection of prioritization techniques. We hypothesized that incorporation of such predictors *would* increase technique effectiveness. We designed an experiment (Experiment 3) to investigate this hypothesis at the function level. The experiment design was analogous to the design used in Experiment 1b except for the addition of eight new techniques: fn-fi-cov-nofb, fn-fi-cov-fb, fn-fi-fep-nofb, fn-fi-fep-fb, fn-diff-cov-nofb, fn-diff-cov-fb, fn-diff-fep-nofb, and fn-diff-fep-fb.

The ANOVA analysis of the data collected in this experiment (see Table 2.9) indicated that these techniques were significantly different. We then followed the same procedure used earlier, employing a Bonferroni analysis to gain insight into the differences. The results are presented in Table 2.10. Three techniques combining FEP and fault proneness (fn-diff-fep-fb, fn-diff-fep-nofb, and fn-fi-fep-nofb) were significantly better than the rest. This suggests that some of the combinations of fault-proneness and FEP estimators we employed did significantly improve the power of our prioritization techniques. Fn-fi-fep-fb and other techniques using ei-

Grouping				Means	Techniques
A				79.479	fn-diff-fep-fb
A				79.450	fn-diff-fep-nofb
A	B			78.712	fn-fi-fep-nofb
	B	C		78.167	fn-fi-fep-fb
		C	D	77.453	fn-fep-fb
		C	D	77.321	fn-fi-cov-nofb
		C	D	77.057	fn-diff-cov-nofb
			D	76.957	fn-fep-nofb
			D	76.928	fn-cov-nofb
E				74.596	fn-fi-cov-fb
E				73.465	fn-cov-fb
F				67.666	fn-diff-cov-fb

TABLE 2.10: Bonferroni Analysis, All Function Level Techniques, All Programs

ther FEP estimates or fault indexes followed. We could not distinguish significant and consistent gains by any particular method (DIFF, FI or FEP) when used individually. Also, the use of feedback seems to have had a negative effect on the techniques using fault proneness as evidenced by the significant superiority of fn-diff-cov-nofb and fn-fi-cov-nofb over fn-diff-cov-fb and fn-fi-cov-fb, respectively.

Table 2.9 shows that the interaction between program and technique was again, in this experiment, significant. So, to better understand the APFD variations, we analyzed the impact of techniques on each program separately. First, we performed univariate ANOVA analysis on each program. The results of those individual ANOVAs were consistent in indicating that all techniques were significantly different. (See Table B.7, in Appendix A.)

We next performed individual Bonferroni analyses per program. These are shown in Tables B.8 and B.9, in Appendix A. As the results show, several programs (`print_tokens`, `print_tokens2`, `tot_info`, and `replace`) exhibited rankings similar to those seen in the overall analysis, though in some cases with fewer significant differences among the techniques. Results on the other programs differed more substantially. On `tcas`, the techniques' APFD values descended gradually, which created a lot of overlap in the top ranked techniques. Still, there was a group of significantly best techniques that included fn-cov-nofb, fn-fi-cov-nofb, fn-cov-fb, and fn-fi-fep-nofb. The techniques using DIFF, however, ranked significantly worse than the others. On `schedule`, in contrast, fn-diff-cov-nofb performed significantly better than the other techniques, and the remaining techniques fell into a series of overlapping groups. A similar picture occurred for `schedule2`, except that here, fn-diff-cov-fb was significantly worse than other techniques. Finally, results on `space` were unique. On this program, techniques using just fault proneness were significantly better than the others. The highest APFD values were generated through fn-fi-cov-fb, which was significantly superior to the other techniques. Combinations of FEP and fault indexes did not work as well as for other programs. Furthermore, the two techniques using just FEP estimates were ranked last.

In summary, on most programs, techniques combining FEP and FI ranked among the top techniques. However, certain programs presented unique characteristics that impacted the effectiveness of those techniques. Still, on all programs, a subset of the techniques using fault proneness measures was considered significantly better than (or not different from) techniques not using that predictor. It is also interesting to note that the use of feedback seemed to have a greater impact on simpler techniques, while on techniques combining FEP and fault proneness

	SS	Degrees of freedom	MS	F	p
PROGRAM	15205111	7	2172159.00	6062.476	0.00
TECHN	4654397	17	273788.00	764.140	0.00
PROGRAM*TECHN	2507689	119	21073.00	58.815	0.00
Error	41709550	116400	358.30		

TABLE 2.11: ANOVA Analysis, All Techniques, All Programs

measures the impact of using feedback did not translate into significant gains (e.g., fn-diff-fep-fb was not significantly different from fn-diff-fep-nofb).

2.4.4.4 Overall Analysis

Finally, to gain an overall perspective on all techniques, we performed ANOVA and Bonferroni analyses on all the techniques including optimal and random (see Tables 2.11 and 2.12). As expected, the ANOVA analysis revealed significant differences among the techniques and the Bonferroni analysis generated groups that confirmed our previous observations. The most obvious observation is that the optimal technique was still significantly better than all other techniques; this suggests that there is still room for improvement in prioritization techniques. However, all techniques significantly outperformed random ordering. St-fep-fb remained the best performing technique after optimal. Yet, the group of techniques ranked next included function level techniques combining fault proneness measures and FEP. These function level techniques were significantly better than st-cov-fb.

Grouping					Means	Techniques
A					94.728	optimal
B					80.733	st-fep-fb
C					79.479	fn-diff-fep-fb
C					79.450	fn-diff-fep-nofb
C	D				78.867	st-fep-nofb
C	D				78.712	fn-fi-fep-nofb
	D	E			78.178	st-cov-nofb
	D	E			78.167	fn-fi-fep-fb
		E	F		77.453	fn-fep-fb
		E	F		77.321	fn-fi-cov-nofb
		E	F	G	77.057	fn-diff-cov-nofb
			F	G	76.957	fn-fep-nofb
			F	G	76.928	fn-cov-nofb
				G	76.077	st-cov-fb
H					74.596	fn-fi-cov-fb
H					73.465	fn-cov-fb
I					67.666	fn-diff-cov-fb
J					62.100	random

TABLE 2.12: Bonferonni Analysis, All Techniques, All Programs

2.4.5 Threats to Validity

In this section we discuss the potential threats to validity of our study, including: (1) threats to internal validity (could other effects on our dependent variables be responsible for our results), (2) threats to construct validity (are our independent variables appropriate), and (3) threats to external validity (to what extent do our results generalize). We also explain how we tried to reduce the chances that those threats affect the validity of our conclusions.

2.4.5.1 *Threats to Internal Validity*

The inferences we made about the effectiveness of the prioritization techniques could have been affected by the following factors. (1) Faults in the prioritization and APFD measurement tools. To control this threat, we performed code reviews on all tools, and validated tool outputs on a small but non-trivial program. (2) Differences in the code to be tested, the locality of program changes, and the composition of the test suite. To reduce this threat, a factorial design was used to apply each prioritization technique to each test suite and each object program. (3) FEP, FI, and DIFF calculations. FEP values are intended to capture the probability, for each test case and each statement, that if the statement contains a fault, the test case will expose that fault. Mutation analysis was used to provide an estimate of these FEP values; however, other estimates might be more precise, and might increase the effectiveness of FEP-based techniques. Similar reasoning applies to our calculations of FI and DIFF.

2.4.5.2 *Threats to Construct Validity*

The goal of prioritization is to maximize some pre-defined criterion by scheduling test cases in a certain order. In this article, we focused on maximizing the rate of fault detection and we defined APFD to represent it. However, APFD is not the only possible measure of rate of fault detection and it has some limitations. (1) APFD assigns no value to subsequent test cases that detect a fault already detected; such inputs may, however, help debuggers isolate the fault, and for that reason might be worth measuring. (2) APFD does not account for the possibility that faults and test cases may have different costs. (3) APFD only partially captures the aspects of the

effectiveness of prioritization; we could also consider other measures for purposes of assessing effectiveness. One might not even want to measure *rate* of detection; one might instead measure the percentage of the test cases in a prioritized test suite that must be run before *all* faults have been detected. (4) We employed a greedy algorithm for obtaining "optimal" orderings. This algorithm may not always find the true optimal ordering, and this might allow some heuristic to actually outperform the optimal and generate outliers. However, a true optimal ordering can only be better than the greedy optimal ordering that we utilized; therefore our approach is conservative, and cannot cause us to claim significant differences between optimal and any heuristic where such significance would not exist.

2.4.5.3 Threats to External Validity

The generalization of our conclusions is constrained by several threats. (1) Object representativeness. The object programs are of small and medium size, and have simple fault patterns that we have manipulated to produce versions with multiple faults. Complex industrial programs with different characteristics may be subject to different cost-benefit tradeoffs. (2) Testing process representativeness. 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 for these two threats can be achieved only through additional studies using a greater range and number of software artifacts.

2.5 Case Studies

In this section we present three case studies which included three medium to large size programs. These case studies offer us the opportunity to scale up our investigation of prioritization techniques by focusing on larger objects drawn from the field.

2.5.1 Experimental Subjects

We considered three programs: `grep`, `flex`, and `QTB`.

2.5.1.1 Grep and Flex

`Grep` and `flex` are common Unix utility programs; `grep` searches input files for a pattern, and `flex` is a lexical analyzer generator. The source code for both programs is publicly available. For this study, we obtained five versions of `grep`, and five of `flex`. The earliest version of `grep` that we used contained 7451 lines of C code and 133 functions; the earliest version of `flex` contained 9153 lines of C code and 140 functions. Tables 2.13 and 2.14 provide data about the numbers of functions and lines changed (modified, added, or deleted) in each of the versions of the two programs, respectively.

The `grep` and `flex` programs possessed the advantage of being publicly available in multiple versions; however, neither program was equipped with test suites or fault data. Therefore, we manufactured these. To do this in as fair and unbiased a manner as possible, we adapted processes used by Hutchins et al. to create the Siemens program materials [93], as follows.

Version	Number of functions changed	Number of lines changed	Number of regression faults
(baseline)	–	–	–
1	81	2488	4
2	40	716	3
3	26	513	3
4	110	1891	1

TABLE 2.13: The `Grep` Object

Version	Number of functions changed	Number of lines changed	Number of regression faults
(baseline)	–	–	–
1	28	333	5
2	72	1649	4
3	12	40	8
4	6	91	1

TABLE 2.14: The `Flex` Object

For each program, we used the category partition method and an implementation of the TSL tool [8, 149] to create a suite of black-box tests, based on the program's documentation. These test suites were created by graduate students experienced in testing, but who were not involved in, and were unaware of, the details of this study. The resulting test suites consisted of 613 test cases for `grep`, exercising 79% of that program's functions, and 525 test cases for `flex`, exercising 89% of that program's functions.

To evaluate the performance of prioritization techniques with respect to rate of detection of regression faults, we require such faults – faults created in a program version as a result of the modifications that produced that version. To obtain such faults for `grep` and `flex`, we asked several graduate and undergraduate computer science students, each with at least two years experience programming in C and each unacquainted with the details of this study, to become familiar with the code of the programs and to insert regression faults into the versions of those programs. 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.

To further direct their efforts, the fault seeders were given the following list of types of faults to consider:

- Faults associated with variables, such as with definitions of variables, redefinitions of variables, deletions of variables, or changes in values of variables in assignment statements.

- Faults associated with control flow, such as addition of new blocks of code, deletions of paths, redefinitions of execution conditions, removal of blocks, changes in order of execution, new calls to external functions, removal of calls to external functions, addition of functions, or deletions of functions.

- Faults associated with memory allocation, such as not freeing allocated memory, failing to initialize memory, or creating erroneous pointers.

After at least twenty potential faults had been seeded in each version of each program,[6] we activated these faults individually, one by one, and executed the test suites for the programs to determine which faults could be revealed by test cases in those suites. We selected, for use in this study, all faults that were exposed by at least one, and at most 20%, of the test cases in the associated test suite. (Exclusion of faults not exposed does not affect APFD results; we chose to exclude faults exposed by more than 20% of the test suites on the grounds that easily-exposed faults are more likely to be detected and removed during testing by developers, and prior to formal regression testing, than faults exposed less easily.) The numbers of faults remaining, and utilized in the studies, are reported in Tables 2.13 and 2.14.

2.5.1.2 QTB

`QTB`[7] is an embedded real-time subsystem that performs initialization tasks on a level-5 RAID storage system. In addition, it provides fault tolerance and recovery capabilities. `QTB` contains over 300K lines of C code combined with hundreds of in-line assembly-code statements across 2875 functions. `QTB` had been under maintenance for several years.

In this study we considered six `QTB` versions, the first of which we considered to be the baseline. Table 2.15 reports details about these versions. The versions con-

---

[6] On version four of `flex`, due to the small number of modifications in that version, fewer than twenty potential faults were initially seeded.

[7] Because the company that created QTB wishes to remain anonymous, we have changed the original names of the subsystem and versions that comprise this object.

Version	Number of functions changed	Total number of regression faults	Number of regression faults exposed by test cases
(baseline)	–	–	–
1	15	10	8
2	3	1	1
3	98	2	2
4	7	2	2
5	169	7	4

TABLE 2.15: The QTB Object

stituted major system releases produced over a six month period. For each version, test engineers employed a regression test suite to exercise system functionalities. The execution of the test suite required, on average, 27 days. The test suite included 135 test cases that exercised 69% of the functions in the baseline version. The coverage information available for QTB is exclusively at the function level. (Software instrumentation tools designed to produce finer granularity coverage data caused the system to fail due to timing problems.)

Maintenance activities applied to QTB resulted in the unintentional incorporation into the system of 22 (discovered) regression faults. Table 2.15 summarizes the fault data. Observe that only 17 of the 22 faults were exposed by the regression test suite across the versions; only these faults factor into the calculation of APFD values. Also, note that the execution of a faulty function did not guarantee exposure of faults in that function.

2.5.2 *Prioritization Techniques*

In this study, we used the following prioritization techniques: function coverage no feedback (fn-cov-nofb), function coverage feedback (fn-cov-fb), function fault index coverage no feedback (fn-fi-cov-nofb), function fault index coverage feedback (fn-fi-cov-fb), function diff coverage no feedback (fn-diff-cov-nofb), function diff coverage feedback (fn-diff-cov-fb), random prioritization, and optimal prioritization. These techniques were described in Section 2.3.

2.5.3 *Case Study Design*

In each case study, we investigate whether some of our previous conclusions on prioritization hold. More precisely, we focus on prioritization techniques at the function level and their ability to improve rate of fault detection. In addition, we explore instances (extreme in some cases) of the techniques' behavior that were not previously visible, which provide us with additional information on their strengths and weaknesses.

Our case studies evaluate prioritization techniques by adapting the "baseline" comparison method described in [53, 107]. This method is meant to compare a newly proposed technique against current practice, which is used as a baseline. In our case studies, assuming that no particular form of prioritization constitutes typical practice, we consider the random technique the baseline against which other techniques are compared.

There is, however, one aspect in which our studies differ from a "typical" baseline study. In our study, although we do not control the evolution of the programs studied, we can execute multiple techniques on the same version of the same pro-

gram. In other words, we are studying programs that evolve naturally, but we can control (and replicate) the execution of prioritization techniques and evaluate their impact based on the data we collected from the evolution of those programs. Still, there are several uncontrolled factors that constrain these studies and the aspects of the problem that we can address. We now proceed to explain the variables involved and the level of control we had over them.

To minimize the misinterpretation of the results that might occur due to specific types or amounts of change in any particular version, we perform our analysis on several versions in each case study. Confounding factors associated with the testing process are not fully controlled. First, we do not control (and do not know) the test generation process employed for QTB. In addition, we have only one test suite in each case study, which may limit our ability to determine whether differences in APFD are due to the techniques or to test suite composition. A similar situation is presented by the faults in the software. Faults were seeded in grep and flex by students not extensively familiar with the application domains, but QTB was used with its original faults. Finally, all the case studies assume that the software development and testing processes remained constant throughout the program evolution.

We investigated eight techniques over each of the units of study. The techniques employed were: random, optimal, fn-cov-nofb, fn-cov-fb, fn-fi-cov-nofb, fn-fi-cov-fb, fn-diff-cov-nofb, fn-diff-cov-fb. (In other words, we used all techniques not involving statement level instrumentation or FEP estimation. We excluded the former because we did not have statement-level coverage information for QTB, and excluded the latter because performing the mutation analysis necessary to estimate FEP for these programs was not feasible.) However, there were two differences involving these techniques due to characteristics of the program data.

First, we obtained the APFD for random by averaging the APFD of 20 random orderings. This differs from the controlled study in which only one ordering per cell was generated. However, in a case study with a much smaller set of observations, we required an "average" random case to avoid extreme instances that could bias our evaluation. Second, the prioritization techniques based on fault proneness that we applied to `QTB` differed slightly from those used in our controlled experiments and our studies on `flex` and `grep`. The `DIFF`-based technique utilized produced just a binary value indicating whether a function changed or not between versions. The FI technique utilized on `QTB` used a subset of the metrics incorporated into the FI metric used in previous experiments. These differences might cause the resulting techniques to be less sensitive to modifications in the versions. Nevertheless, for simplicity and despite these differences, in this study we continue to use the nomenclature used to denote these techniques in earlier studies.

2.5.4 Results and Analysis

Figure 2.5 provides an overview of the data for the three case studies. We include two graphs for each of the programs studied; these graphs provide complementary information. The box plots on the left present the overall distribution of APFD data per technique, summarized across all versions. This depiction illustrates each techniques' mean and variation, allowing comparisons of overall performance across all versions. The graphs on the right present the APFD values achieved by each of

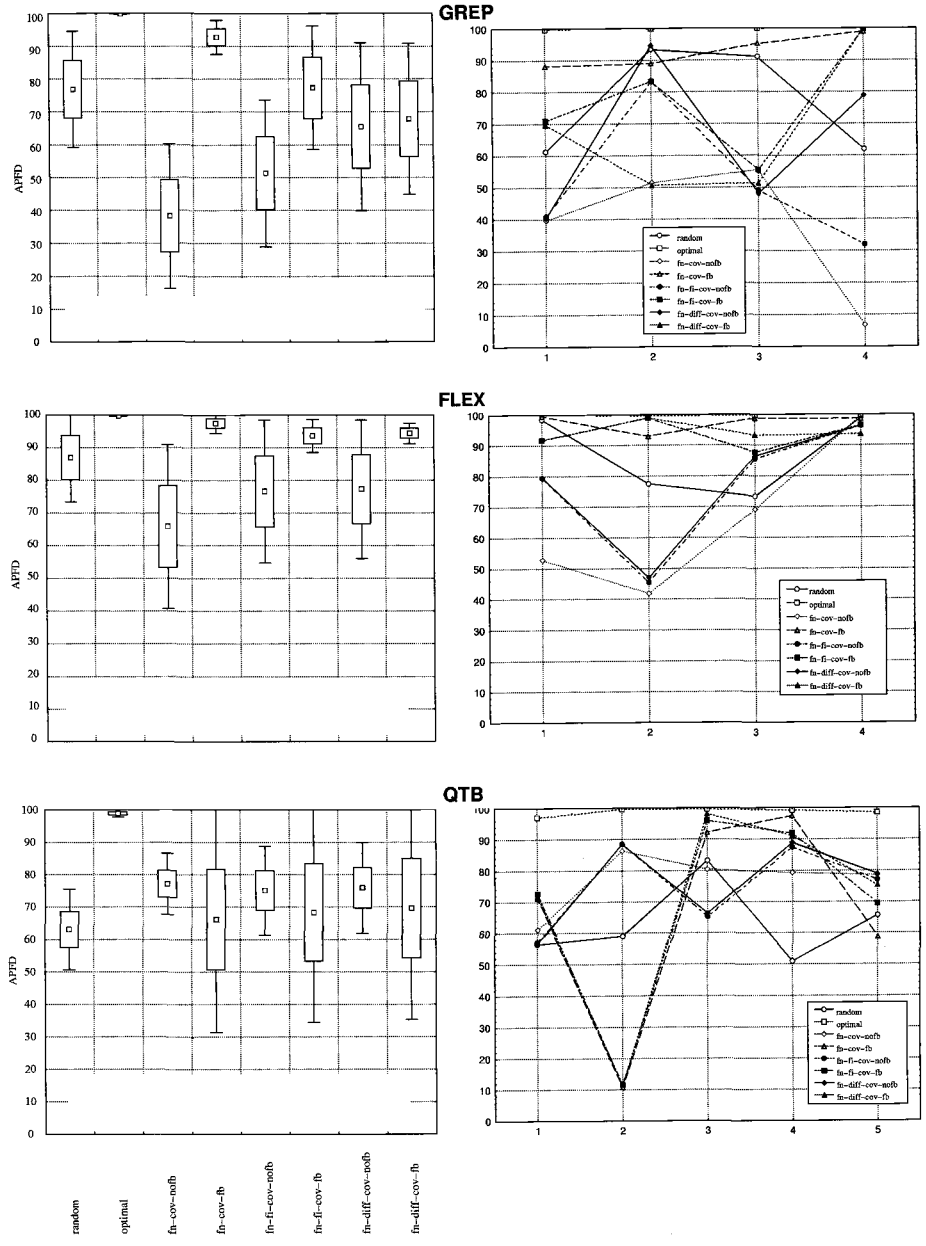


FIGURE 2.5: Overview of case study data. Vertical axes depict APFD values. At left, box plots present the overall distribution of APFD data per technique, summarized across all program versions. At right, graphs show the APFD values obtained by each technique on each version.

the techniques across each of the versions, allowing comparisons on a per version basis.[8]

We consider overall results (box plots) first. On both `grep` and `flex`, in terms of mean APFD, optimal ranks first, fn-cov-fb ranks second, and fn-fi-cov-fb ranks third. On both programs, techniques using feedback (fb) produce APFDs closer to optimal than do techniques not using feedback (nofb). On `QTB`, in contrast, the average APFD for techniques not using feedback exceeds the average APFD for techniques using feedback. Further, on `grep` and `flex`, techniques using feedback exhibited less variance in APFD than those not using feedback, whereas on `QTB` this relationship was reversed. Another surprise was the high mean APFD values exhibited by the random technique on `grep` and `flex`. On `QTB`, the random technique outperforms the other techniques in some cases (evident in the extents of the tails of the distributions), but in terms of mean APFD it is the worst performing technique overall.

The data presented in the graphs of per version results, Figure 2.5, also contains several surprises. It seems that the primary constant across different programs is the high degree of change in APFD values across versions. Furthermore, from the figures, it is difficult to understand the "contradictions" that are present in the data. However, when each specific scenario is analyzed in detail, a clearer picture emerges.

We conjecture that the variability in the results observed in these case studies can be attributed, at least in part, to the location of faults in the program and the

[8] Each technique has one value for each version within each program. These values have been connected with lines to facilitate the visualization of patterns.

Program	Version	Faulty functions	Exposed faults	Pct. of functions executed by test cases		Pct. of functions executed by fault exposing test cases		Pct. of test cases executing faulty functions		Pct. of test cases exposing faults	
				Mean	Stdev	Mean	Stdev	Mean	Stdev	Mean	Stdev
grep	1	4	4	44.22	5.25	41.89	4.25	64.19	46.75	0.25	0.1
	2	2	3	45.64	5.27	42.68	3.77	93.31	8.30	3.37	2.21
	3	3	3	46.82	5.39	45.07	5.65	85.80	23.60	2.29	0.93
	4	1	1	46.80	5.39	39.38	0.48	99.02	0.00	0.33	0
flex	1	4	5	52.52	6.20	59.39	4.46	73.33	38.73	13.29	5.46
	2	4	4	58.37	6.50	63.18	12.38	49.81	57.08	1.29	1.46
	3	6	7	58.37	6.50	66.61	4.85	52.77	51.66	7.48	8.55
	4	1	1	58.37	6.50	72.04	3.60	98.48	0.00	17.52	0
QTB	1	10	8	22.91	10.24	22.99	10.77	74.35	32.42	1.02	0.38
	2	1	1	22.91	10.24	37.57	0.00	97.78	0.00	0.74	0
	3	2	2	22.91	10.24	27.97	6.94	36.67	48.71	1.11	0.52
	4	2	1	22.91	10.24	36.45	10.37	33.33	0.00	1.11	0.52
	5	7	4	22.91	10.24	34.19	7.49	48.64	31.98	1.11	0.43

TABLE 2.16: Fault Exposure and Test Case Activity Data

likelihood that those faults are executed by the test cases in the test suite. (This conjecture is based, in part, on results presented in [193].) We observe that fault location and test coverage patterns varied widely across our programs and versions, and this may have contributed to the variability in our results. To investigate this conjecture, we need to understand those factors within each unit of study. Table 2.16 summarizes relations between faults, fault exposure, and test coverage for each of the programs and versions studied, listing data on the percentage of functions executed by test cases, the percentage of functions executed by fault exposing test cases, the percentage of test cases executing faulty functions, and the percentage of test cases exposing faults.

First, we consider why techniques using feedback did not perform (overall) as well on QTB as on the other two programs. Observe on the per version graph for

QTB that techniques using feedback performed better than those not using feedback on versions 1, 3, and 4, slightly worse on version 5, and considerably worse on version 2. With only five versions, the influence of one poor performance (version 2) is sufficient to affect the overall rankings of means exhibited across all versions of the program.

To suggest why version 2 exhibited such results, we turn to the data in Table 2.16. As the table shows, on version 2 of QTB, 98% of the test cases for the system execute the faulty function, but only one of those test cases exposes the fault. Consider also version 4 of grep. Here, as on version 2 of QTB, most of the program's test cases (99.02%) execute the faulty function, and few of these test cases (only two, or 0.33% of the test suite) expose the fault. Despite this similarity, however, these two cases result in different relationships between techniques using and not using feedback.

This difference may be attributed to differences in test case execution patterns. On version 2 of QTB, the test cases exposing the fault execute a larger percentage of functions (37.57%) than the average test case (22.91%). On version 4 of grep, in contrast, the test cases exposing the fault execute a smaller percentage of functions (39.38%) than the average test case (46.8%).

When test cases that expose faults execute a relatively small percentage of functions, they are likely to be scheduled near the end of test execution by techniques not using feedback (e.g., fn-cov-nofb). For faults exposed by a small percentage of the test cases that reach them, the postponing of such test cases further postpones the exposure of those faults, exacerbating the differences in APFD values achieved by the prioritization techniques.

Summarizing, characteristics involving the coverage achieved by the test suite and the location of the faults affect the results of prioritization techniques using and not using feedback. In our case studies, where each version has significantly different types and locations of faults, and test execution patterns with relationship to those faults differ widely, the tests exposing faults change and so does the effectiveness of the techniques across versions.

Next, we consider the situations in which random performs better than some of our prioritization heuristics, by considering differences in the relationship between random and fn-cov-nofb on versions 1 and 4 of `flex`. (On version 1, random outperforms fn-cov-nofb; on version 4 the two are nearly equivalent).

Intuitively, random could be expected to perform well when the chances of exposing a fault with an arbitrary test case are high. Versions 1 and 4 of `flex` reflect this expectation. On both of these versions (for which over 13.29% and 17.52% of the test cases, respectively, expose faults), random prioritization produces APFD values relatively close to the optimal values. On version 4 this means that it is likely that one of the first six test cases randomly selected will expose the fault in that version. On version 4, however, a relatively large percentage of functions (72.83%) are executed by fault exposing test cases, and most test cases (98.48%) execute faulty functions, rendering it probable that fn-cov-nofb will also perform well. On version 1, in contrast, a smaller percentage of functions (59.76%) are executed by fault exposing test cases, and fewer test cases (73.33%) execute faulty functions. In this case, the probability that fn-cov-nofb will postpone execution of fault exposing functions is increased; a random rule thus performs better.

Similar cases can be seen on versions 2 and 3 of `grep`. On the other hand, when faulty functions are not likely to be executed and faults are not likely to be exposed

by arbitrary test cases, the random technique performs poorly. For example, on version 1 of `grep`, the likelihood of exposing a fault is very small (0.25%) so random performed poorly. A similar situation can be found on version 3 of `flex`, where some faults have a very small probability of being exposed (as suggested by the high standard deviation).

Finally, we were surprised that techniques using fault proneness estimates did not provide more substantial improvements. Although in many specific instances, incorporation of fault proneness estimates added significant improvements to techniques, the mean APFDs of these techniques across versions is not favorable. A previous study [42] of the FI fault index supported our expectations for techniques using fault proneness estimates; however, that previous study evaluated the predictive abilities of fault proneness indexes, whereas the study reported here evaluates techniques that employ those indexes to schedule test cases. In addition, there are other factors, such as test exposure capability, program domain, particular processes, and technique scalability, that may have not been relevant in the earlier studies, but could have had a significant impact on the fault prediction procedure [110] and in the FI-based prioritization techniques' effectiveness. These limitations might be contributing to some of the differences that we observe across the case studies.[9]

This said, FI-based techniques were observed to result in improved APFD values in our controlled experiments; thus, the difference in results exhibited in these

---

[9] Repeatability problems such as this, where different studies yield different results, are not unique to testing. For example, Lanubile et al. [110] report that even successful fault proneness prediction models might not work on every data set, and that there is a need to take into consideration the context in which they are used.

case studies is of interest. We can suggest at least three things that may explain these differences. First, in our controlled experiments, the ratio between the amount of code changed and the number of faults in the program versions utilized was much smaller than in these case studies. The number of lines of code changed in the controlled experiments numbered in the tens, while the average number of changes in the case studies numbered in the hundreds. Since "regression" fault proneness metrics associate evolutionary changes with fault likelihood, they are likely to be more effective when fewer changes are made. Second, the test suites used in the case studies had different characteristics than those used in the controlled experiments. We have suggested ways in which test suite characteristics can impact techniques using and not using feedback differently, and the same suggestions apply to techniques employing measures of fault proneness. Third, the fault seeding process used on some objects (small Siemens programs, `grep`, and `flex`) could have played a role in the variance we observed among the techniques' performance, especially on the techniques based on fault proneness. Although we attempted to perform this process as consistently as possible, we recognize that it constitutes an artificial procedure that might not provide an accurate reflection of reality.

It is important to note that the previous interpretations are not always as transparent as presented. For example, in the presence of multiple faults, some of which are exposed by a large number of test cases and some of which are infrequently exposed, interpretation becomes more difficult and the results less predictable. Nevertheless, these general patterns can be observed repeatedly. It can also be observed that version specific prioritization can (and most often does) yield considerable gains over random test case orderings and, if we select the proper prioritization technique, those gains can be maximized.

2.6 Conclusions

As we saw from our experiments, change-based prioritization was effective in improving the rate of fault detection in our small Siemens program subjects. It would be interesting to compare statement-level change-based techniques to function-level ones. While statement-level techniques were effective, their application can be limited for reasons discussed in Section 2.1. Statement-level fep-based techniques were especially effective; however, because fep estimation is so expensive, these techniques may not be practical for the large software systems. As for function-level techniques, change-based and, especially, change/fep combination-based techniques were most effective. These may be more affordable than statement-level fep-based techniques. Function-level techniques did lose effectiveness relative to statement-level ones; however, these losses were marginal relative to the variations between the best function-level and statement-level techniques.

The situation was different for the medium subjects. Here, change-based techniques did not show significant improvement over coverage-based ones. This difference can be explained in terms of the way in which faults were distributed. In the Siemens subjects, faults are changes, meaning that every changed location has an equal number of faults in it. Under these circumstances, there is a strong correlation between the amount of change in a given function and the number of faults it contains. In our medium subjects, faults were artificially seeded in modified code. The number of faults was relatively small and only a subset of changed functions contained faults. In this case, the correlation between the amount of change in a given function and the number of faults that it contains is not very strong, resulting in limited effectiveness of change-based techniques.

We might hypothesize that for real-world programs, change-based techniques can be sufficiently effective. The reason for this is that large software systems typically have larger numbers of faults in them. In each release, many changes may be applied. Because faults occur naturally in this case, their density would be similar in all changes. If this is not the case, fault index will capture the amount of changes and the complexity of the code, estimating the fault existence probability in a given function. As a result, the effectiveness of change-based techniques could be expected to be substantial relative to other techniques.

The most important conclusion is that change-based prioritization techniques can be effective and, simultaneously, very inexpensive to employ in regression testing.

Applying prioritization can add costs due to the analysis needed to obtain needed data. An important question arises as to whether a difference in APFD values makes any practical impact on the cost of regression testing, and whether this impact would be large enough to overcome prioritization cost making prioritization cost-effective. We study this question in detail in Chapter 5.

CHAPTER 3

COST COGNIZANCE

3.1 Introduction

The prioritization techniques presented in the previous chapter, like most earlier published ones, assume that all test cases are equally expensive and all faults are equally severe.[1] While this is appropriate in some cases, in other cases it is an over-simplification.[2] Some test cases can simply detect an error in input and terminate almost immediately, while other test cases can involve computations that require hours to complete. Similarly, in some cases, a test case requires usage of resources such as equipment, expendable materials, or human labor, while a different test case may use little or no equipment or human labor. Under these scenarios, when evaluating the relative worth of test cases, we need to take into account these differences in costs.

Similarly, in many cases, faults can differ in cost. One fault can be a simple spelling error in an interface which many users would tolerate. On the other hand, another fault can result in incorrect parameters supplied to a device, which can result in program failure, or even in catastrophes such as loss of aircraft control

---

[1] Wong [213] outlines a prioritization algorithm that incorporates data on test costs.

[2] Portions of this chapter have appeared previously in [47].

or radiation overdose. Fault severity, too, may be an important component of test worth.

To address these issues, in this chapter, we present several *cost-cognizant* prioritization techniques that take into account varying test costs and fault severities. We extend the $APFD$ metric to incorporate variable test costs and fault severities. Finally, we perform an empirical study, exploring costs, severities, ways to estimate them, and cost-cognizant prioritization techniques.

3.2 Cost Cognizant Measure

3.2.1 Limitations of the APFD Metric

As just stated, the APFD metric presented in Section 1.2.5 relies on two assumptions: (1) all faults have equal severity, and (2) all test cases have equal costs. These assumptions are manifested in the fact that the metric simply plots the percentage of faults detected against the fraction of the test suite run. To consider possible effects of these assumptions, we present several simple examples.

Example 1. Consider the testing scenario illustrated in Figure 1.1 in Chapter 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 1.1, 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 severities 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 1.1, 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 1.1.D). Consider alternative test case order **E–C–B–A–D**. This order is illustrated with respect to the APFD metric in Figure 3.1; 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-

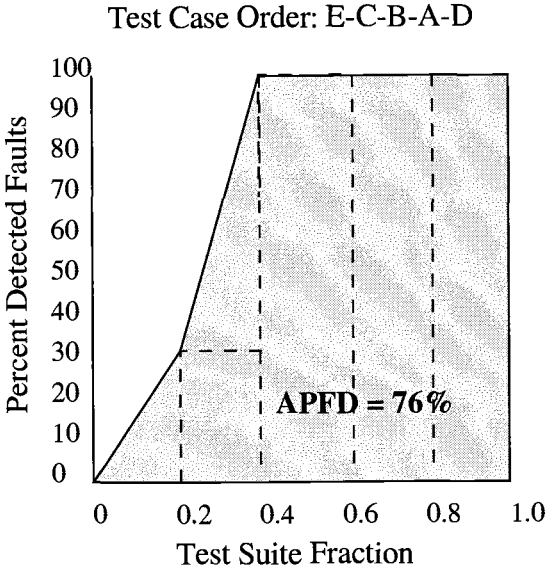


FIGURE 3.1: APFD for Example 3.

detected-per-hour than the first order up through the end of execution of the second test case.

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

Example 4. Finally, consider an example in which both fault severities 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 severities 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**.[3] Again, the APFD metric would not make this distinction.

3.2.2 A New Cost-cognizant Metric

3.2.2.1 Examples and Motivation

The foregoing examples suggest that, when considering the relative merits of test cases, a measure for rate-of-fault-detection that assumes that test case costs and fault severities 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, and testers face and make decisions about this tradeoff frequently. It is thus appropriate that this tradeoff be considered when prioritizing test cases. A metric for evaluating test case orders must accommodate the factors underlying this tradeoff. It is our thesis that such a metric should *reward test case orders proportionally to their rate of "units-of-fault-severity-detected-per-unit-test-cost"*.

We have created such a metric by adapting our APFD metric; we call our new "cost-cognizant" metric $APFD_C$. In terms of the graphs used in Figures 1.1 and 3.1, creation of this new metric entails two modifications. First, instead of letting the horizontal axis in such a graph denote "Test Suite Fraction", we let it denote

---

[3] In this example we have assumed, for simplicity, that a "unit" of fault severity is 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. We discuss this further in the following sections.

"Percentage Total Test Case Cost Incurred". Now, each test case 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, instead of letting the vertical axis in such a graph denote "Percent Detected Faults", we let it denote "Percentage Total Fault Severity Detected". 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 severity that fault accounts for.

In this context, the "cost" of a test case and the "severity" of a fault can be interpreted and measured in various ways. If time (for test execution, setup, and validation) is the primary component of test case cost then execution time can suffice to measure that cost. However, practitioners could also assign test case costs based on factors such as hardware costs or engineers' salaries.

Similarly, fault severity could be measured strictly in terms of the time required to locate and correct a fault, or practitioners could factor in the costs of lost business, litigation, damage to persons or property, and so forth. In any case, the $APFD_C$ metric supports these interpretations. (Note that in the context of the $APFD_C$ metric we are concerned not with predicting these costs, which may be difficult, but with measuring the costs after they have occurred, in order to evaluate various test case orders.)

Under this new interpretation, in graphs such as those just discussed, a test case's contribution is "weighted" along the horizontal dimension in terms of test case cost, and along the vertical dimension in terms of the cumulative severity of faults it reveals. In such graphs, the curve delimits a greater area for a test case order that exhibits greater "units-of-fault-severity-detected-per-unit-test-cost"; the area delimited constitutes our new $APFD_C$ metric.

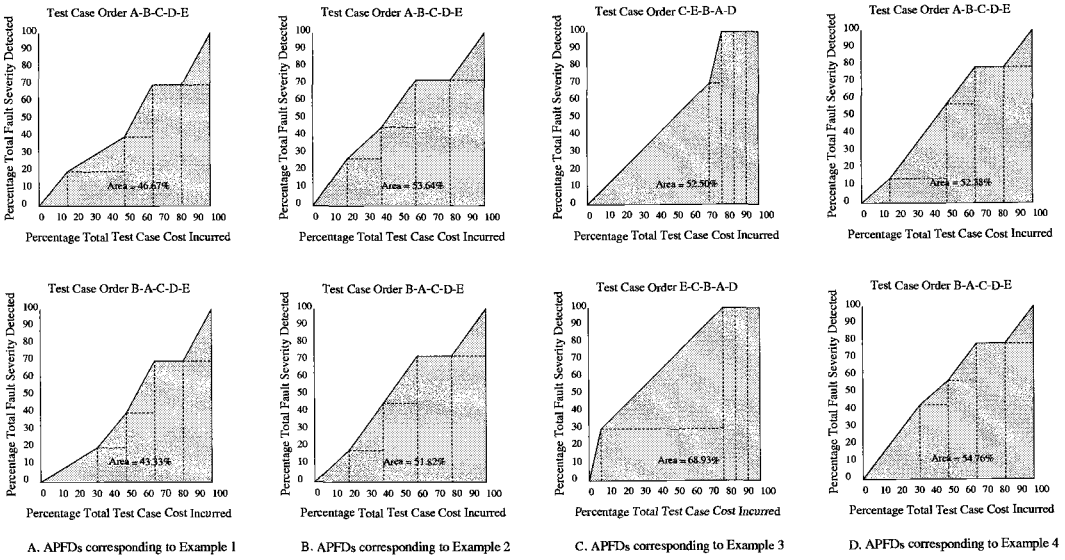


FIGURE 3.2: Examples illustrating the APFD_C metric.

To illustrate the APFD_C metric from this graphical point of view, Figure 3.2 presents graphs for each of the four examples presented in Section 3.2.1. The left-most pair of graphs (Figure 3.2.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 faster-detecting order, **A–B–C–D–E**.

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 3.2.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 severity value assigned to faults 2-10 is 1 and the severity value assigned to fault 1 is 2. The pair of graphs in Figure 3.2.C, corresponding to Example 3, show how the

new metric distinguishes test case orders involving a high-cost test case **C**: instead of under-valuating 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 3.2.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 severity 3 while all other faults have severity 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**.

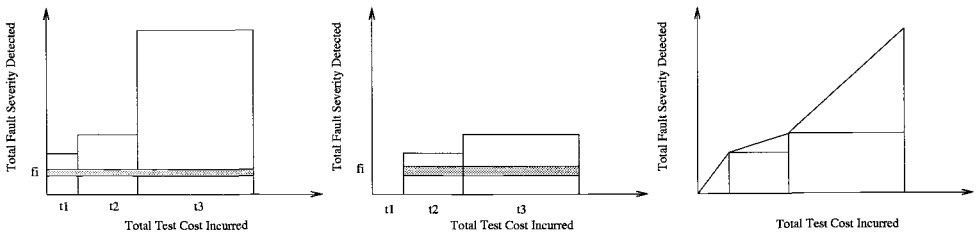
3.2.2.2 Derivation of the Formula for the $APFD_C$ Metric

Let T be a test suite containing n test cases with costs $t_1, t_2, \ldots, t_n$. Let F be a set of m faults revealed by T, and let $f_1, f_2, \ldots, f_m$ be the severities of those faults. Let TF_i be the first test case in an ordering T' of T that reveals fault i.

We first derive the formula for the area under the graph in Figure 3.3, and then normalize it to obtain a value between 0 and 1.

Consider two graphs, Figures 3.3(a) (upper graph) and 3.3(b) (lower graph). It is easy to see that the area we wish to compute is the average of the areas under these graphs. Each graph can be represented as set of slices (shown as shaded bars on the graphs), corresponding to each fault. The slice for fault i starts from the test case that reveals i first under the given order. In the graph from Figure 3.3(b), this slice begins at the TF_i-th test case and extends to the end (n-th test case). In the graph from Figure 3.3(a), this slice begins at the $(TF_i - 1)$-th test case and extends to the end (n-th test case).

The area of the slice for the upper graph is $f_i \times \sum_{j=TF_i}^{n} t_j$.



(a) upper graph (b) lower graph (c) target average graph

FIGURE 3.3: Graphs for use in illustrating derivation of the $APFD_C$ formula.

The area of the slice for the lower graph is $f_i \times \sum_{j=TF_i+1}^{n} t_j$ (for the fault detected by the last test case, it is zero).

The area of the upper graph is $\sum_{i=1}^{m}(f_i \times \sum_{j=TF_i}^{n} t_j)$.

The area of the lower graph is $\sum_{i=1}^{m}(f_i \times \sum_{j=TF_i+1}^{n} t_j)$.

The average of the areas of the upper and lower graphs is

$$\begin{aligned}
&\frac{1}{2}\left(\sum_{i=1}^{m}\left(f_i \times \sum_{j=TF_i}^{n} t_j\right)+\sum_{i=1}^{m}\left(f_i \times \sum_{j=TF_i+1}^{n} t_j\right)\right)= \\
&\qquad \frac{1}{2}\left(\sum_{i=1}^{m}\left(f_i \times \left(\sum_{j=TF_i}^{n} t_j + \sum_{j=TF_i+1}^{n} t_j\right)\right)\right)= \\
&\qquad \frac{1}{2}\left(\sum_{i=1}^{m}\left(f_i \times \left(\sum_{j=TF_i}^{n} t_j + \sum_{j=TF_i}^{n} t_j - t_{TF_i}\right)\right)\right)= \\
&\qquad \frac{1}{2}\left(\sum_{i=1}^{m}\left(f_i \times \left(2\sum_{j=TF_i}^{n} t_j - t_{TF_i}\right)\right)\right)= \\
&\qquad \sum_{i=1}^{m}\left(f_i \times \left(\sum_{j=TF_i}^{n} t_j - \frac{1}{2}t_{TF_i}\right)\right)
\end{aligned} \tag{3.1}$$

The normalized graph area ($APFD_C$ value) is

$$\frac{\sum_{i=1}^{m}(f_i \times (\sum_{j=TF_i}^{n} t_j - \frac{1}{2}t_{TF_i}))}{\sum_{i=1}^{m} f_i \times \sum_{j=1}^{n} t_j} \tag{3.2}$$

The (cost-cognizant) weighted average percentage of faults detected during the execution of test suite T' is thus given by the equation:

$$\text{APFD}_C = \frac{\sum_{i=1}^{m} (f_i \times (\sum_{j=TF_i}^{n} t_j - \frac{1}{2}t_{TF_i}))}{\sum_{i=1}^{n} t_i \times \sum_{i=1}^{m} f_i} \tag{3.3}$$

If all test case costs are identical and all fault costs are identical ($\forall i\ t_i = t$ and $\forall i\ f_i = f$), this formula simplifies as follows:

$$\frac{\sum_{i=1}^{m} (f \times (\sum_{j=TF_i}^{n} t - t\frac{1}{2}))}{n \times m \times t \times f} =$$
$$\frac{\sum_{i=1}^{m} (n - TF_i + 1 - \frac{1}{2})}{nm} =$$
$$1 - \frac{TF_1 + TF_2 + ... + TF_m}{nm} + \frac{1}{2n}$$

Thus, Equation 3.3 remains applicable when either test case costs or fault severities are identical, and when both test case costs and fault severities are identical the formula reduces to the formula for APFD.

3.3 Estimating Test Cost

There are two issues connected with test cost: measuring or estimating if for use in assessing an order in terms of $APFD_C$, and estimating if for use in prioritizing test cases.

The cost of a test case indicates the resources involved in executing and validating that test case. Various objective measures are possible, for example: when the primary cost resource is machine or human time, test cost can measure the actual

time needed to test the software with a given test case. Another measure considers the monetary costs of test case execution and validation; this may reflect amortized hardware cost, wages, cost of materials required for testing, earnings lost due to delays in failing to meet the target release date, and so on.

Test cost is relatively easy to obtain postfactum, for use in $APFD_C$ computation. Here we need to observe what resources each test case required. Cost estimation is more difficult to do before testing starts, however, which is necessary if we wish to use the data in prioritization. In this case, we need to predict test costs. One approach for doing this is to analyze a test case and the code it invokes in order to estimate test cost. Another approach, and the one that we use, is to rely on test cost assessments made during the previous testing session. Under this approach, we assume that test costs do not change significantly from one version to another.

3.4 Estimating Fault Severity

As with test cost, with fault severity there are two issues: measuring or estimating it for use in assessing an order in terms of $APFD_C$, and estimating it for use in prioritizing test cases.

The severity of a fault is related to the costs or resources required if a fault persists in, and influences, the software after it is released. Various measures of these severities are possible; for example:

- One approach measures the severity of a fault as the amount of money lost due to the fault. This approach could apply to software where faults can bring disastrous outcomes resulting in losses that include litigation, human life, equipment, and so forth.

- A second approach measures a fault's effects on software reliability. This approach applies to software where faults result in inconvenience to users, and a single failure is not likely to have safety-critical effects (e.g., office productivity software). In this case, the main effect involves decreased reliability which could result in loss of customers.

As with test costs, we are concerned with both before- and after-the-fact fault severity estimation. When testing completes, we have information about faults and can estimate their severities to use in $APFD_C$ computation (though this is not as easy as with test cost). On the other hand, before testing begins, we need to estimate the severity of potential faults as related to our system or tests, so this information can be used by prioritization, and this is far more difficult than with test cost. To be precise, we require a metric on test cases that correlates satisfactorily with the severity of the faults given test cases reveal.

If we knew the faults revealed by a given test case, and knew their severities, correlating test cases with severities would be trivial. In practice, however, this information is not available before testing is completed. Two possible estimation approaches, however, involve assessing *module criticality* and *test criticality*. In assessing module criticality, we attempt to relate fault severity to the criticality of a module (or some other code component such as a block, function, file, or object) in which a fault may occur. In assessing test criticality, we attempt to relate test cases directly to the severity of faults they may detect. By estimating module or test criticality, we hope to incorporate fault severity into test case prioritization before testing begins. The empirical study reported later in this chapter examines the first approach further.

3.5 Prioritization Techniques

In this section, we present prioritization techniques adapted to accommodate estimations of non-uniform test costs and estimations of fault severities.

We extend Algorithm 1, presented in Section 2.3, to provide techniques that include cost cognizance. The cost-cognizant entities include test costs, test criticalities, and location/module criticalities.

In the change-based techniques described in Chapter 1.4, test award values are computed by summing products of coverage information and other data. To incorporate module criticality, we multiply module criticality into the corresponding products and sum them as before. Test costs and test criticalities are incorporated as ratios of test criticality over cost. These ratios can then be used in various ways: (1) multiply these ratios by the award values of corresponding tests (*multiplication*), (2) sort tests by award values first and then resolve ties using criticality-over-cost ratio (*award first*), and (3) sort by ratios first and resolve ties by award values (*ratio first*).

The foregoing techniques, especially change-based ones, can result in large numbers of ties in award values. Thus, additional tie resolution may be needed. For example, if the multiplication combination function is used, ties can be resolved in terms of simple change-based award values. Alternatively, total coverage (no feedback) can be used to resolve ties.

Our cost-cognizant techniques are represented by the generalized algorithm presented as Algorithm 2. In order to instantiate this generalized algorithm as a particular technique, proper parameters must be provided to the algorithm. These

Algorithm 2 Cost-cognizant test case prioritization.

Procedure **Prioritize**
In: $_{\text{orig}}coverage$ (original location-wise binary coverage),
$locations$ (set of program locations),
$tests$ (set of test cases),
fep (fault-exposing-potential),
$change$ (change information),
$feedback$ (parameter specifying whether feedback is to be used),
$mcrit$ (module criticality),
$tcost$ (test cost),
$tcrit$ (test criticality)
$update\_conf$ (parameter specifying whether to update confidence vector),
$award\_term\_func$ (parameter function which computes a single term for award value)
$need\_to\_reset$ (parameter function that computes whether a reset is needed)
Out: $list$

$list = \epsilon$
for all $l \in locations$ **do**
  for all $t \in tests$ **do**
    $coverage_{l,t} = {}_{orig}coverage_{l,t}$
  end for
  $confidence_l = 0$
end for
while $tests \neq \emptyset$ **do**
  $awards$ = **Compute_Award_Values**($_{\text{orig}}coverage$, $coverage$, $change$, fep, $confidence$, $mcrit$, $tcost$, $tcrit$, $tests$, $locations$, $award\_term\_func$)
  if need_to_reset($awards$, $tests$) = 1 **then**
    $coverage$ = **Reset**($_{\text{orig}}coverage$, $locations$, $tests$)
    $awards$ = **Compute_Award_Values**($_{\text{orig}}coverage$, $coverage$, $change$, fep, $confidence$, $mcrit$, $tcost$, $tcrit$, $tests$, $locations$, $award\_term\_func$)
  end if
  find the test $t_{\text{selected}} \in tests$ with the largest award value $awards_{t_{\text{selected}}}$
  add t_{selected} to $list$
  remove t_{selected} from $tests$
  if $feedback = true$ **then**
    $coverage$ = **Update_Coverage**($_{\text{orig}}coverage$, $coverage$, $locations$, $tests$, t_{selected})
    if $update\_conf = true$ **then**
      $confidence$ = **Update_Confidence**($_{\text{orig}}coverage$, $fep_{l,t_{\text{selected}}}$, $confidence$, $locations$, $tests$, t_{selected})
    end if
  end if
end while
return $list$

Algorithm 2 Cost-cognizant test case prioritization (Continued).

Procedure **Update_Coverage**
In: $_{\text{orig}}coverage$, $coverage$, $tests$, $locations$, t_{selected}
Out: $newcoverage$

for all $l \in locations$ **do**
  for all $t \in tests$ **do**
    if $_{\text{orig}}coverage_{l,t_{\text{selected}}} = 1$ **then**
      $newcoverage_{l,t} = 0$
    else
      $newcoverage_{l,t} = coverage_{l,t}$
    end if
  end for
end for
return $coverage$

Procedure **Update_Confidence**
In: $_{\text{orig}}coverage$, fep, $confidence$, $tests$, $locations$, t_{selected}
Out: $newconfidence$

for all $l \in locations$ **do**
  if $_{\text{orig}}coverage_{l,t_{\text{selected}}} = 1$ **then**
    $newconfidence_l = 1 - (1 - confidence_l) \times (1 - fep_{l,t_{\text{selected}}})$
  else
    $newconfidence_l = confidence_l$
  end if
end for
return $newconfidence$

Procedure **Compute_Award_Values**
In: $_{\text{orig}}coverage$ (original binary coverage information),
  $coverage$ (current original binary coverage information),
  $change$ (change information),
  fep (fault-exposing-potential),
  $confidence$ (confidence vector),
  $mcrit$ (module criticality),
  $tcost$ (test cost),
  $tcrit$ (test criticality),
  $tests$ (set of tests),
  $locations$ (set of locations),
  $award\_term\_func$ (function to compute terms in award value)
Out: $awards$ (test award values)

for all $t \in tests$ **do**
  $awards_t =< 0, ..., 0 >$
  for all $l \in locations$ **do**
    $awards_t = awards_t +$ award_term_funcs($_{\text{orig}}coverage_{l,t}, coverage_{l,t}, change_l, fep_{l,t}, mcrit_l, tcrit_t, tcost_t$)
  end for
end for
return $awards$

Procedure **Reset**
In: $_{\text{orig}}coverage$, $locations$, $tests$
Out: $coverage$

for all $t \in tests$ **do**
  for all $l \in locations$ **do**
    $coverage_{l,t} = {}_{\text{orig}}coverage_{l,t}$
  end for
end for
return $coverage$

necessary parameters are described in Table 3.1, which lists the various techniques expressed by Algorithm 2.

The main difference between Algorithm 2 and the one presented in Chapter 1.4 (Algorithm 1) is incorporation of costs and criticalities. Module criticalities, test criticalities, and test costs are incorporated into the award value computation in the three previously described ways.

The parameter, $award\_term\_func$, supplies a function used in award value computation for computing a single term. This function can be any of the functions $FUNC1$ to $FUNC12$, given in Figures 3.4 to 3.7.

The parameter, $need\_to\_reset$, supplies a function that indicates whether data structures need to be reset. This function can be any of the functions: $NTR1$ to $NTR4$, given in Figure 3.8.

In Table 3.1, we include techniques that use change information. However, as for Algorithm 1 in Chapter 1.4, there are four types of change information, resulting in four different classes of change-based techniques: fi-based, binary fi-based, diff-based, and binary diff-based. Thus, each technique in the table that uses change information represents a set of four techniques, one per change information type. So, for all techniques whose mnemonic contains, in its second field, "mod", this field can be any of the following: "fi", "bfi", "diff", or "bdiff".

If certain criticality or cost data are missing, unit values should be substituted in place of the missing data.

technique	locations	change	feed-back	update-_conf	award-_term_func	need_to-_reset
st-cov-ccmult-nofb	statement	unit	false	false	FUNC1	NTR1
st-cov-ccmult-fb	statement	unit	true	false	FUNC1	NTR1
st-cov-cca1st-nofb	statement	unit	false	false	FUNC2	NTR1
st-cov-cca1st-fb	statement	unit	true	false	FUNC2	NTR1
st-cov-ccr1st-nofb	statement	unit	false	false	FUNC3	NTR2
st-cov-ccr1st-fb	statement	unit	true	false	FUNC3	NTR2
fn-cov-ccmult-nofb	function	unit	false	false	FUNC1	NTR1
fn-cov-ccmult-fb	function	unit	true	false	FUNC1	NTR1
fn-cov-cca1st-nofb	function	unit	false	false	FUNC2	NTR1
fn-cov-cca1st-fb	function	unit	true	false	FUNC2	NTR1
fn-cov-ccr1st-nofb	function	unit	false	false	FUNC3	NTR2
fn-cov-ccr1st-fb	function	unit	true	false	FUNC3	NTR2
st-fep-ccmult-nofb	statement	unit	false	false	FUNC4	NTR1
st-fep-ccmult-fb	statement	unit	true	true	FUNC4	NTR1
st-fep-cca1st-nofb	statement	unit	false	false	FUNC5	NTR1
st-fep-cca1st-fb	statement	unit	true	true	FUNC5	NTR1
st-fep-ccr1st-nofb	statement	unit	false	false	FUNC6	NTR2
st-fep-ccr1st-fb	statement	unit	true	true	FUNC6	NTR2
fn-fep-ccmult-nofb	function	unit	false	false	FUNC4	NTR1
fn-fep-ccmult-fb	function	unit	true	true	FUNC4	NTR1
fn-fep-cca1st-nofb	function	unit	false	false	FUNC5	NTR1
fn-fep-cca1st-fb	function	unit	true	true	FUNC5	NTR1
fn-fep-ccr1st-nofb	function	unit	false	false	FUNC6	NTR2
fn-fep-ccr1st-fb	function	unit	true	true	FUNC6	NTR2
fn-mod-cov-ccmult-nofb	function	change	false	false	FUNC1	NTR1
fn-mod-cov-ccmult-fb	function	change	true	false	FUNC1	NTR1
fn-mod-cov-cca1st-nofb	function	change	false	false	FUNC2	NTR1
fn-mod-cov-cca1st-fb	function	change	true	false	FUNC2	NTR1
fn-mod-cov-ccr1st-nofb	function	change	false	false	FUNC3	NTR2
fn-mod-cov-ccr1st-fb	function	change	true	false	FUNC3	NTR2
fn-mod-fepm-ccmult-nofb	function	change	false	false	FUNC7	NTR1
fn-mod-fepm-ccmult-fb	function	change	true	false	FUNC7	NTR1
fn-mod-fepm-cca1st-nofb	function	change	false	false	FUNC8	NTR1
fn-mod-fepm-cca1st-fb	function	change	true	false	FUNC8	NTR1
fn-mod-fepm-ccr1st-nofb	function	change	false	false	FUNC9	NTR2
fn-mod-fepm-ccr1st-fb	function	change	true	false	FUNC9	NTR2
fn-mod-fep-ccmult-nofb	function	change	false	false	FUNC10	NTR2
fn-mod-fep-ccmult-fb	function	change	true	false	FUNC10	NTR2
fn-mod-fep-cca1st-nofb	function	change	false	false	FUNC11	NTR2
fn-mod-fep-cca1st-fb	function	change	true	false	FUNC11	NTR2
fn-mod-fep-ccr1st-nofb	function	change	false	false	FUNC12	NTR3
fn-mod-fep-ccr1st-fb	function	change	true	false	FUNC12	NTR3

TABLE 3.1: Prioritization Techniques and Parameters for use with Algorithm 2

Function **FUNC1**
In: $_{\text{orig}}coverage\_term$ (original coverage data for a given location and test),
$coverage\_term$ (current coverage data for a given location and test),
$change\_term$ (change data for a given location)
fep_term (fep data for a given location and test)
$confidence\_term$ (current confidence data for a given location)
$mcrit\_term$ (module criticality data for a given location)
$tcrit\_term$ (test criticality data)
$tcost\_term$ (test cost data)
Out: $term$

$term1 = mcrit\_term \times coverage\_term \times change\_term$
$term2 = {}_{\text{orig}}coverage\_term$
$ratio = tcrit\_term \times tcost\_term$
$term =< term1 \times ratio, term1, term2 >$
return $term$

Function **FUNC2**
In: $_{\text{orig}}coverage\_term$ (original coverage data for a given location and test),
$coverage\_term$ (current coverage data for a given location and test),
$change\_term$ (change data for a given location)
fep_term (fep data for a given location and test)
$confidence\_term$ (current confidence data for a given location)
$mcrit\_term$ (module criticality data for a given location)
$tcrit\_term$ (test criticality data)
$tcost\_term$ (test cost data)
Out: $term$

$term1 = mcrit\_term \times coverage\_term \times change\_term$
$term2 = {}_{\text{orig}}coverage\_term$
$ratio = tcrit\_term \times tcost\_term$
$term =< term1, ratio, term2 >$
return $term$

Function **FUNC3**
In: $_{\text{orig}}coverage\_term$ (original coverage data for a given location and test),
$coverage\_term$ (current coverage data for a given location and test),
$change\_term$ (change data for a given location)
fep_term (fep data for a given location and test)
$confidence\_term$ (current confidence data for a given location)
$mcrit\_term$ (module criticality data for a given location)
$tcrit\_term$ (test criticality data)
$tcost\_term$ (test cost data)
Out: $term$

$term1 = mcrit\_term \times coverage\_term \times change\_term$
$term2 = {}_{\text{orig}}coverage\_term$
$ratio = tcrit\_term \times tcost\_term$
$term =< ratio, term1, term2 >$
return $term$

FIGURE 3.4: Specific award value term computing functions, used as parameters in Algorithm 2.

Function **FUNC4**
In: ${}_{\text{orig}}coverage\_term$ (original coverage data for a given location and test),
$coverage\_term$ (current coverage data for a given location and test),
$change\_term$ (change data for a given location)
fep_term (fep data for a given location and test)
$confidence\_term$ (current confidence data for a given location)
$mcrit\_term$ (module criticality data for a given location)
$tcrit\_term$ (test criticality data)
$tcost\_term$ (test cost data)
Out: $term$

$term1 = mcrit\_term \times {}^{\text{orig}}coverage\_term \times (1 - confidence\_term) \times fep\_term$
$term2 = {}_{\text{orig}}coverage\_term$
$ratio = tcrit\_term \times tcost\_term$
$term =< term1 \times ratio, term2 >$
return $term$

Function **FUNC5**
In: ${}_{\text{orig}}coverage\_term$ (original coverage data for a given location and test),
$coverage\_term$ (current coverage data for a given location and test),
$change\_term$ (change data for a given location)
fep_term (fep data for a given location and test)
$confidence\_term$ (current confidence data for a given location)
$mcrit\_term$ (module criticality data for a given location)
$tcrit\_term$ (test criticality data)
$tcost\_term$ (test cost data)
Out: $term$

$term1 = mcrit\_term \times {}^{\text{orig}}coverage\_term \times (1 - confidence\_term) \times fep\_term$
$term2 = {}_{\text{orig}}coverage\_term$
$ratio = tcrit\_term \times tcost\_term$
$term =< term1, ratio, term2 >$
return $term$

Function **FUNC6**
In: ${}_{\text{orig}}coverage\_term$ (original coverage data for a given location and test),
$coverage\_term$ (current coverage data for a given location and test),
$change\_term$ (change data for a given location)
fep_term (fep data for a given location and test)
$confidence\_term$ (current confidence data for a given location)
$mcrit\_term$ (module criticality data for a given location)
$tcrit\_term$ (test criticality data)
$tcost\_term$ (test cost data)
Out: $term$

$term1 = mcrit\_term \times {}^{\text{orig}}coverage\_term \times (1 - confidence\_term) \times fep\_term$
$term2 = {}_{\text{orig}}coverage\_term$
$ratio = tcrit\_term \times tcost\_term$
$term =< ratio, term1, term2 >$
return $term$

FIGURE 3.5: Specific award value term computing functions, used as parameters in Algorithm 2.

Function **FUNC7**
In: $_{orig}coverage\_term$ (original coverage data for a given location and test),
$coverage\_term$ (current coverage data for a given location and test),
$change\_term$ (change data for a given location)
fep_term (fep data for a given location and test)
$confidence\_term$ (current confidence data for a given location)
$mcrit\_term$ (module criticality data for a given location)
$tcrit\_term$ (test criticality data)
$tcost\_term$ (test cost data)
Out: $term$

$term1 = mcrit\_term \times coverage\_term \times change\_term \times (1 - confidence\_term) \times fep\_term$
$term2 =$ $_{orig}coverage\_term$
$ratio = tcrit\_term \times tcost\_term$
$term =< term1 \times ratio, term2 >$
return $term$

Function **FUNC8**
In: $_{orig}coverage\_term$ (original coverage data for a given location and test),
$coverage\_term$ (current coverage data for a given location and test),
$change\_term$ (change data for a given location)
fep_term (fep data for a given location and test)
$confidence\_term$ (current confidence data for a given location)
$mcrit\_term$ (module criticality data for a given location)
$tcrit\_term$ (test criticality data)
$tcost\_term$ (test cost data)
Out: $term$

$term1 = mcrit\_term \times coverage\_term \times change\_term \times (1 - confidence\_term) \times fep\_term$
$term2 =$ $_{orig}coverage\_term$
$ratio = tcrit\_term \times tcost\_term$
$term =< term1, ratio, term2 >$
return $term$

Function **FUNC9**
In: $_{orig}coverage\_term$ (original coverage data for a given location and test),
$coverage\_term$ (current coverage data for a given location and test),
$change\_term$ (change data for a given location)
fep_term (fep data for a given location and test)
$confidence\_term$ (current confidence data for a given location)
$mcrit\_term$ (module criticality data for a given location)
$tcrit\_term$ (test criticality data)
$tcost\_term$ (test cost data)
Out: $term$

$term1 = mcrit\_term \times coverage\_term \times change\_term \times (1 - confidence\_term) \times fep\_term$
$term2 =$ $_{orig}coverage\_term$
$ratio = tcrit\_term \times tcost\_term$
$term =< ratio, term1, term2 >$
return $term$

FIGURE 3.6: Specific award value term computing functions, used as parameters in Algorithm 2.

Function **FUNC10**
In: $_{\text{orig}}coverage\_term$ (original coverage data for a given location and test),
$coverage\_term$ (current coverage data for a given location and test),
$change\_term$ (change data for a given location)
fep_term (fep data for a given location and test)
$confidence\_term$ (current confidence data for a given location)
$mcrit\_term$ (module criticality data for a given location)
$tcrit\_term$ (test criticality data)
$tcost\_term$ (test cost data)
Out: $term$

$term1 = mcrit\_term \times coverage\_term \times change\_term$
$term2 = mcrit\_term \times {}^{\text{orig}}coverage\_term \times (1 - confidence\_term) \times fep\_term$
$term3 = {}_{\text{orig}}coverage\_term$
$ratio = tcrit\_term \times tcost\_term$
$term =< term1 \times ratio, term2 \times ratio, term3 >$
return $term$

Function **FUNC11**
In: $_{\text{orig}}coverage\_term$ (original coverage data for a given location and test),
$coverage\_term$ (current coverage data for a given location and test),
$change\_term$ (change data for a given location)
fep_term (fep data for a given location and test)
$confidence\_term$ (current confidence data for a given location)
$mcrit\_term$ (module criticality data for a given location)
$tcrit\_term$ (test criticality data)
$tcost\_term$ (test cost data)
Out: $term$

$term1 = mcrit\_term \times coverage\_term \times change\_term$
$term2 = mcrit\_term \times {}^{\text{orig}}coverage\_term \times (1 - confidence\_term) \times fep\_term$
$term3 = {}_{\text{orig}}coverage\_term$
$ratio = tcrit\_term \times tcost\_term$
$term =< term1, term2, ratio, term3 >$
return $term$

Function **FUNC12**
In: $_{\text{orig}}coverage\_term$ (original coverage data for a given location and test),
$coverage\_term$ (current coverage data for a given location and test),
$change\_term$ (change data for a given location)
fep_term (fep data for a given location and test)
$confidence\_term$ (current confidence data for a given location)
$mcrit\_term$ (module criticality data for a given location)
$tcrit\_term$ (test criticality data)
$tcost\_term$ (test cost data)
Out: $term$

$term1 = mcrit\_term \times coverage\_term \times change\_term$
$term2 = mcrit\_term \times {}^{\text{orig}}coverage\_term \times (1 - confidence\_term) \times fep\_term$
$term3 = {}_{\text{orig}}coverage\_term$
$ratio = tcrit\_term \times tcost\_term$
$term =< ratio, term1, term2, term3 >$
return $term$

FIGURE 3.7: Specific award value term computing functions, used as parameters in Algorithm 2.

Procedure **NTR1**
In: $awards, tests$
Out: $decision$

$decision = 0$
if first elements of $awards$ are all zeroes **then**
  $decision = 1$
end if
return $decision$

Procedure **NTR2**
In: $awards, tests$
Out: $decision$

$decision = 0$
if second elements of $awards$ are all zeroes **then**
  $decision = 1$
end if
return $decision$

Procedure **NTR3**
In: $awards, tests$
Out: $decision$

$decision = 0$
if first and second elements of $awards$ are all zeroes **then**
  $decision = 1$
end if
return $decision$

Procedure **NTR4**
In: $awards, tests$
Out: $decision$

$decision = 0$
if second and third elements of $awards$ are all zeroes **then**
  $decision = 1$
end if
return $decision$

FIGURE 3.8: Specific need to reset functions, used as parameters in Algorithm 2.

3.6 Case Study

3.6.1 Introduction

To investigate the practical application of our metric and some of the ramifications of using it, we conducted a case study. The goal of the case study was to investigate how different test case cost and fault severity distributions can affect the rate of fault detection as measured by $APFD_C$.

The notion of module criticality was used to estimate fault costs for usage in prioritization. The study examined the effects of various test cost, fault severity, and combined test cost and fault severity distributions on the relative effectiveness of prioritization techniques.

3.6.1.1 Subject

We used the *space* program described in Section 2.4.2.

3.6.1.2 Prioritization Techniques

We selected five prioritization techniques: function coverage cost-cognizant multiply feedback (fn-cov-ccmult-fb), statement coverage cost-cognizant multiply feedback (st-cov-ccmult-fb), function fault index coverage cost-cognizant multiply feedback (fn-fi-cov-ccmult-fb), and random prioritization.

3.6.1.3 *Test Case Cost Distributions*

We randomly assigned costs to tests based on five distributions:

- *Unit:* all test case costs are ones. This corresponds to the case in which test case costs are not considered.
- *Random:* test case costs are uniformly distributed over the range $[1 \ldots 10]$.
- *Normal:* test costs are normally distributed over the range $[1 \ldots 10]$, with mean 5.0, standard deviation 2.0.
- *Mozilla:* test costs are distributed as in the Mozilla application (see Table 3.2) into four levels. Mozilla was the original name for Netscape Communicator and is now an open-source web browser involving hundreds of developers and thousands of testers.[4]
- *QTB:* test costs are distributed as in the QTB application (see Table 3.3). QTB is a real time embedded software system of more than 300KLOC (see Section 2.5.1 and Reference [42] for details).

To apply each test case cost distribution (other than unit, whose application was trivial) we generated a set of cost numbers having the required distribution, and then randomly mapped those numbers to test cases.

---

[4] For more information on Mozilla see www.mozilla.org. For more information on Mozilla testing and fault recording see bugzilla.mozilla.org.

Name	level	description	percentage
HTML	1	least expensive	87
Printing	2	more expensive	1
Smoke tests	3	expensive	2
Buster	4	most expensive	10

TABLE 3.2: Mozilla Test Case Cost Distribution

level	description	percentage
1	not expensive	88
10	expensive	12

TABLE 3.3: QTB Test Case Cost Distribution

3.6.1.4 Fault Severity Distributions

We used three fault severity distributions, as follows:

- *Unit:* all fault severities are ones. This corresponds to the case in which fault severities are not considered.

- *Mozilla linear:* fault severities are distributed as in the Mozilla application (see Table 3.4), into six levels; these costs are assigned on a linear scale from 1 through 6.[5]

[5] This distribution was obtained by querying the bugzilla database for "Resolved" bugs on the Netscape Browser for PC-windows2000.

linear level	exponential level	severity	percentage
1	1	trivial	2
2	2	minor	11
3	4	normal	6
4	8	major	76
5	16	critical	4
6	32	blocking	2

TABLE 3.4: Mozilla Fault Severity Distributions

- *Mozilla exponential:* similar to Mozilla linear, but with fault severity values assigned on an exponential scale from 2^0 through 2^5.

Applying the unit fault severity distribution was trivial. Applying the two Mozilla fault severity distributions, however, was more difficult. The difficulty is that our prioritization techniques use information on module criticality in an attempt to prioritize in a manner that accounts for severity, but we did not have any historical data to support the estimation of module criticality. Thus, we required the generation of both module criticality and fault severity assignments.

Creating these two assignments independently cannot reflect any correlation between module criticalities and fault severities, and the existence of such a correlation is a prerequisite for prioritization techniques that use module criticality to predict fault severity. Instead, our approach was to assume that a correlation between module criticalities and fault severities exists, and rely on this assumption in generating module criticality and fault severity assignments. To apply each fault severity distribution (other than unit) we generated a set of severity numbers with the required distribution, and randomly mapped those numbers to modules. We then considered

	unit	random	normal	Mozilla	QTB
unit	X	X	X	X	X
Moz. linear	X			X	
Moz. exponential	X			X	

TABLE 3.5: Fault Severity Distributions (Left) versus Test Case Cost Distributions (Above). Entries Marked with "X" Indicate Combinations that were Utilized in the Study.

each fault f, and assigned to f a severity number equal to the criticality number of the module containing f.

This approach does not allow us to investigate and compare prioritization techniques fairly, unless our investigations are restricted to conditional hypotheses beginning with the clause: "Suppose the assumption that a close correlation between module criticality and fault severity exists is valid". However, our focus in this study was not on the performance of prioritization techniques, but rather, on the effects that fault distributions and choices of severity values can have on APFD_C. In presenting our data, we condition our conclusions to reflect this methodology.

3.6.1.5 Combinations of Test Cost and Fault Severity Distributions

Given these five test case cost distributions and three fault severity distributions, there were fifteen combinations of test case cost and fault distributions to consider. We restricted our attention to nine of these, as shown in Table 3.5.

3.6.2 *Results and Discussion*

We present the results of our study in three steps. First, we analyze the impact of varying test case cost distributions across different prioritization techniques while maintaining a unit fault severity distribution. Second, we describe the effects of varying fault severity distributions across different prioritization techniques while maintaining a unit test case cost distribution. Third, we combine and analyze the effects of non-unitary distributions for both test case cost and fault severity.

3.6.2.1 *Varying Test Case Cost Distributions*

Figure 3.9 provides an initial view of the $APFD_C$ values measured under different test case cost distributions in our study. The figure displays five sets of bars – one set presenting $APFD_C$ values averaged across all runs with all techniques (left), and one set per technique presenting $APFD_C$ values averaged across all runs with that technique. Each set of bars contains five individual bars – one for each test cost distribution studied. The height of each bar denotes the average $APFD_C$ measured for test suites prioritized under the technique and distribution associated with that bar.[6]

As the figure shows, in our study, test case cost distribution did have an impact on the overall average rate of fault detection of prioritized test suites as measured by $APFD_C$, across all runs with all techniques (as shown by the leftmost set of bars). These differences were statistically significant; however, they were not as large as

[6] The complete statistical analysis of our data can be found in Appendix B. The complete analysis includes the results of applying ANOVA analysis and Tukey tests to all combinations of techniques and cost distributions.

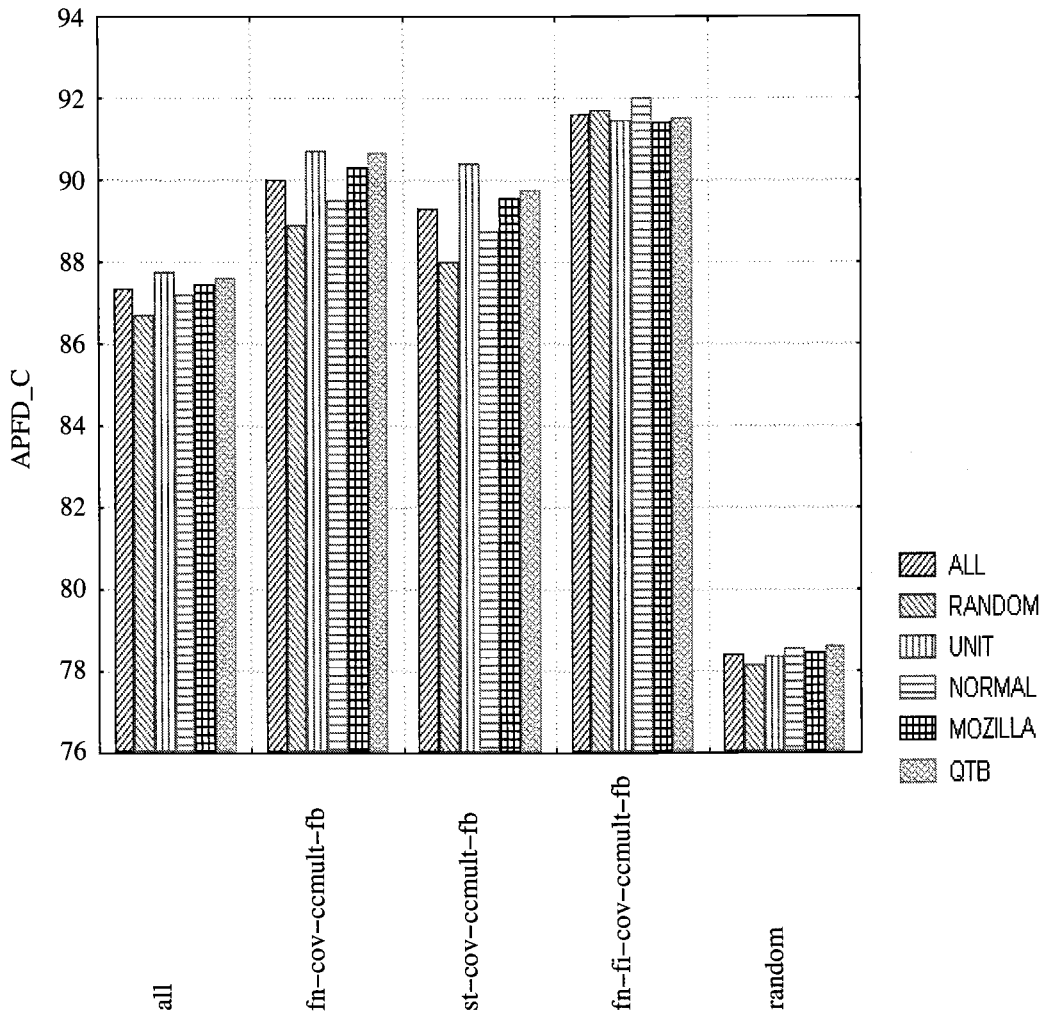


FIGURE 3.9: Mean $APFD_C$ per distribution, per technique.

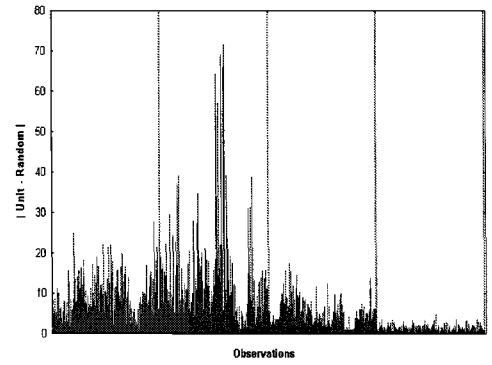
we expected: the average overall $APFD_C$ values for the different distributions differing by no more than one percentage point. Moreover, discriminating by technique (within each of the four rightmost sets of bars) we discovered that the extent of the impact varied with technique. For example, under st-cov-ccmult-fb, the variance in

average $APFD_C$ across different distributions was statistically significant, whereas under fn-fi-cov-ccmult-fb it was not.

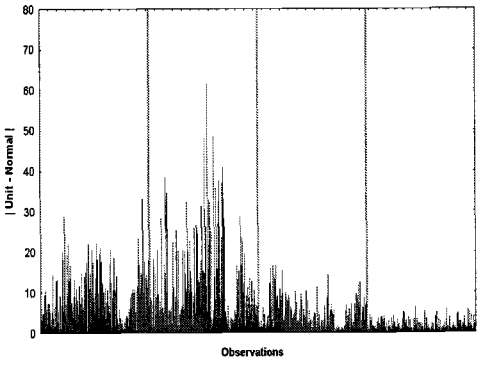
The preceding analysis, however, was performed on average $APFD_C$ values, and an investigation of individual differences in $APFD_C$ yields a different picture. The graphs in Figure 3.10 illustrate the *absolute differences in $APFD_C$ values* of prioritized test suites under the unit test case cost distribution compared to the four other distributions (subfigures A through D, respectively). In each graph, the horizontal axis plots 2000 $APFD_C$ observations: one for each of the 50 prioritized suites under each of the ten versions, for each of the four prioritization techniques. The observations are sorted by technique in order fn-cov-ccmult-fb, st-cov-ccmult-fb, fn-fi-cov-ccmult-fb, random, and within technique by test suite and version. (The solid vertical lines in the figures delimit the boundaries between the data points for the four techniques.)

Figure 3.10 illustrates the extent to which, for individual prioritized suites, $APFD_C$ values under each distribution differ from $APFD_C$ values obtained with unit test case costs. In a few cases, the differences in $APFD_C$ exceed 50%, and in many cases they exceed 20%. This provides further evidence of the need to include test case cost distribution as an integral part of $APFD_C$ (failing to do so may provide poor prioritization). Further, because the unit test case cost distribution produces $APFD_C$ values equivalent to those produced by the (original) APFD metric, these differences show the extent to which the APFD and $APFD_C$ metrics differ.

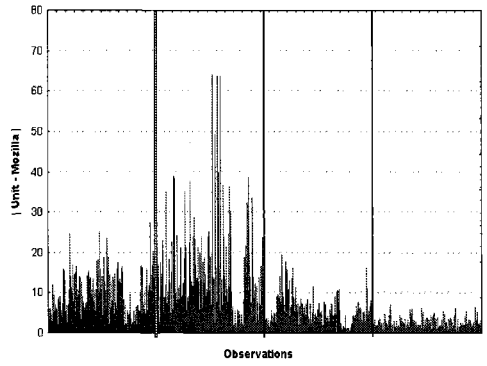
Figure 3.10 also shows that under fn-cov-ccmult-fb and st-cov-ccmult-fb, test cost distributions exhibited higher variability in $APFD_C$ than under the other prioritization techniques (with st-cov-ccmult-fb exhibiting the greatest variability).



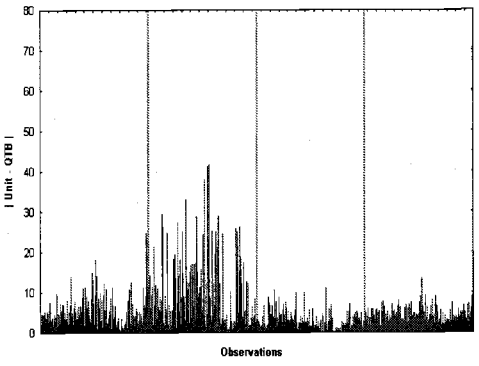
A. Absolute differences: Unit vs Random



B. Absolute differences: Unit vs Normal



C. Absolute differences: Unit vs Mozilla



D. Absolute differences: Unit vs QTB

FIGURE 3.10: Absolute differences in $APFD_C$ values across all observations, for three non-unit distributions vs. the unit distribution.

This suggests that for certain techniques the behavior of the distributions is more predictable than for others.

Finally, returning to Figure 3.9, all prioritization techniques provided significant improvements in $APFD_C$ over the random technique on average, under all test case cost distributions. Thus, even for a technique that is not the "best", and independent of test case cost distribution, prioritization improved rate of fault detection.

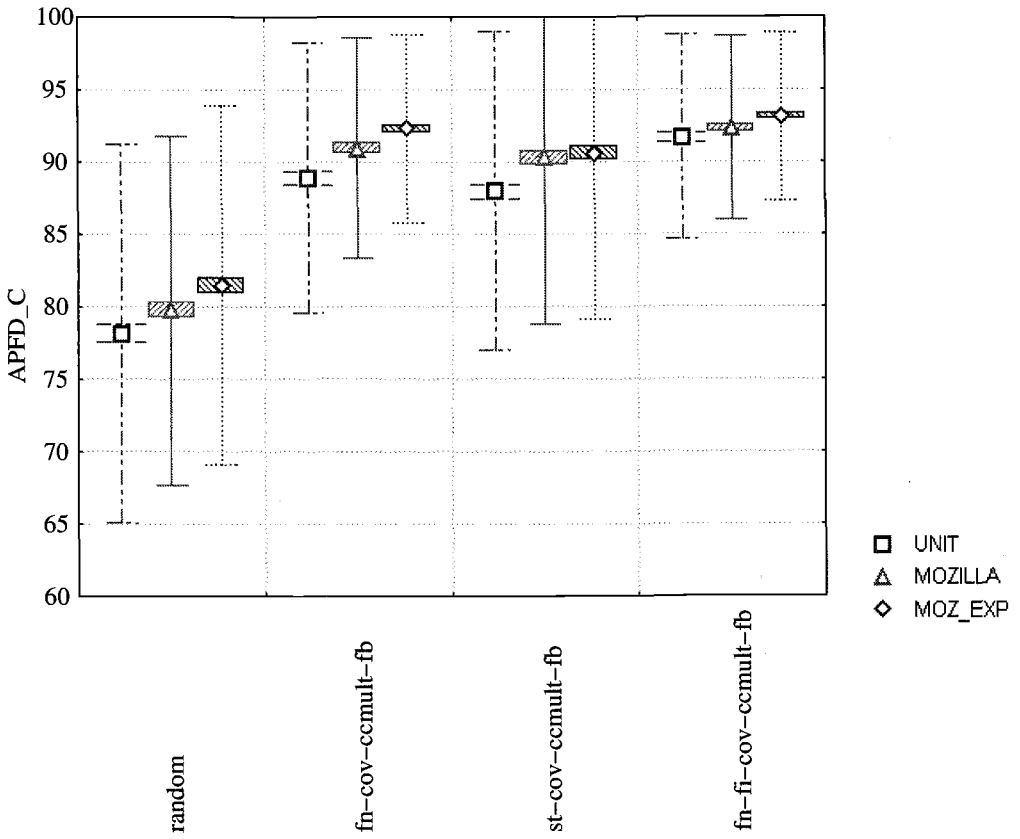


FIGURE 3.11: $APFD_C$ values, per distribution, per technique.

3.6.2.2 Varying Fault Severity Distributions

Our analysis of fault severity distributions shows that fault severity distribution had a significant effect on the $APFD_C$ values for each prioritization technique. Figure 3.11 illustrates. The figure displays three sets of box plots (one set for each technique); each set of box plots contains one box plot for each of the three fault severity distributions. Each individual box plot shows the distribution of $APFD_C$ values over the test suites prioritized by its associated technique and its fault severity distribution.

As the figure shows, fn-fi-cov-ccmult-fb exhibited the most consistent behavior across distributions; it also exhibited better performance than other techniques. Random, in contrast, was the most susceptible to variation across distributions, and exhibited the poorest performance among the techniques.

3.6.2.3 Varying Test Cost and Fault Severity Distributions

Having analyzed the effects of varying test case cost and fault severity distributions individually, we now consider the results obtained by varying both distributions concurrently. To simplify the presentation, we focus on the behavior of $APFD_C$ under just 50 randomly sampled cases of st-cov-ccmult-fb.

Figure 3.12 presents a scatter plot to represent three combined distributions: (1) unit test case cost with unit fault severity, (2) Mozilla test case cost with linear Mozilla fault severity, (3) Mozilla test case cost with exponential Mozilla fault severity. The x-axis represents the $APFD_C$ value under the unit-unit distribution, and each point plotted represents the value of $APFD_C$ under one of the other two distributions.

The plot shows that the $APFD_C$ values for the unit-unit distribution were significantly different from the $APFD_C$ values under the other distributions; this is evident from the large variance in the plots for both Mozilla distributions. The choice of different combinations of test case cost and fault severity distributions did thus have an impact on $APFD_C$.

To further illustrate the differences among the distributions we fitted the individual cases with linear equations using regression analysis, and drew 95% confidences intervals around the regression lines. In the figure, the central line for each distri-

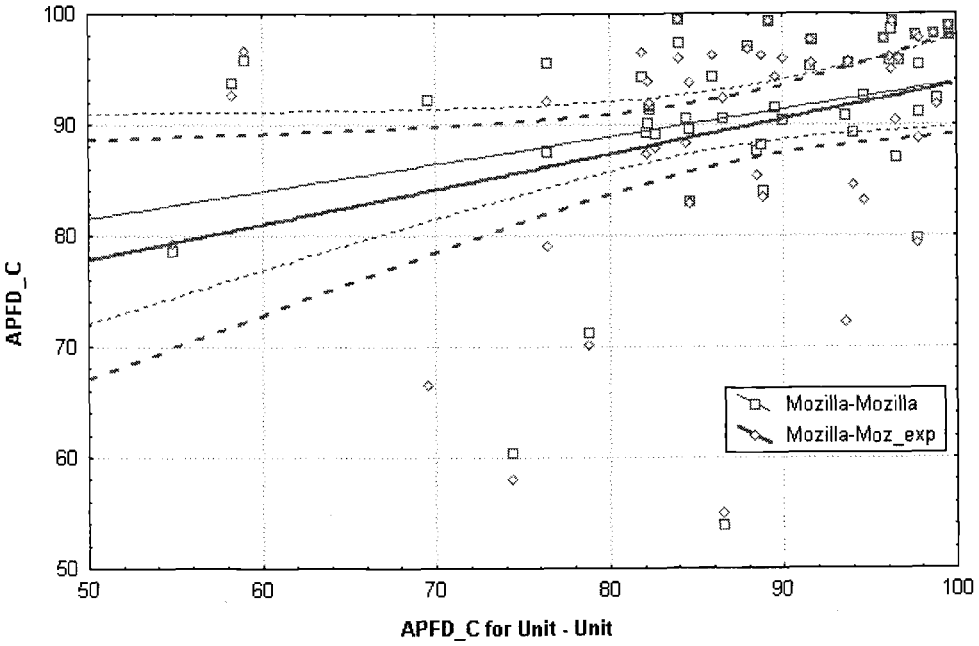


FIGURE 3.12: $APFD_C$ values for combined distributions.

bution is the regression line for that distribution; the other two lines (bands) for each distribution represent the probability that the "true" fitted line (within the population of points considered) falls between the bands. The lines suggest that the Mozilla/Mozilla-exponential distribution combination provides the highest $APFD_C$ values, but as $APFD_C$ becomes closer to 100, both of the distribution combinations involving Mozilla converge.

3.6.3 *Discussion*

To begin to explore ramifications of these results we consider several practical questions test engineers might have, related to the results, and show how we can go about answering them.

Practical Question 1. I am about to develop a new test suite for one of our products. What can I do to make the test suite more suitable for prioritization?

We conjecture that the use of many smaller test cases, rather than fewer larger ones, provides opportunities for test case scheduling, increasing the potential for prioritization gains. The Unit distribution with all small test case costs (and the Mozilla and QTB distributions to a lesser extent), for which average APFD_C was higher than those of other distributions, provides initial evidence to support this conjecture. By partitioning larger test cases, the scope of the average test case decreases, allowing prioritization techniques to discriminate more precisely between test cases. Expensive test cases, even in small numbers, can limit the opportunities for prioritization (this is more likely when expensive test cases cover sections of the program that are likely to contain faults).

Practical Question 2. I have access to two fault severity distributions A and B corresponding to previous releases of our product. I performed a small study using the two distributions, and found that distribution B provided a higher $APFD_C$ than distribution A on our previous release, but the difference in $APFD_C$s was not large. There is no difference in the costs of using the different distributions. How can I determine which distribution to use?

Even a small difference in APFD_C can indicate that a particular distribution can find faults that may cause the most damage earlier in a testing cycle. Whether

this difference will amount to a *practical* difference, however, depends on other factors. For example, if the test suite executes quickly it may make no sense to use prioritization in the first place. To answer the question, an illustration of the practical impact of the differences between $APFD_C$ values is needed.

One approach is to construct the graphs representing the $APFD_C$ scores for the two distributions on the previous release. Then, alter the horizontal axis label to reflect actual test case execution costs (e.g., substituting time). From such a graph it will be more clear how the two orders compare.

To illustrate, we performed such an analysis on one version of `space` under the fn-fi-cov-ccmult-fb technique. In this case, the $APFD_C$ for unit test case cost with unit fault severity was 80%, the $APFD_C$ for unit test case cost with Mozilla linear fault severity was 84%, and the APFD for unit test case cost with Mozilla exponential fault severity was 93%.

The graph in Figure 3.13 depicts the tops of the $APFD_C$ curves for these 3 distributions, with the x-axis representing the percentage of total test case costs incurred and the y-axis representing the percentage of total fault severity detected. After having incurred only 2% of the test case costs, the differences between distributions has become evident. The Mozilla-exponential fault severity distribution has captured 80% of the accumulated fault severity, the Mozilla-linear fault severity distribution has captured approximately 50%, and the unit fault severity distribution has captured less than 35%. The disparity among distributions grows smaller as additional test cost is incurred, and by the time 40% of the test cost has been accounted for, 100% of the fault severity has been captured by all three distributions.

What is the practical impact of these differences? Suppose first that the scale below the x-axis represents the time in days that are required for the test suite to be

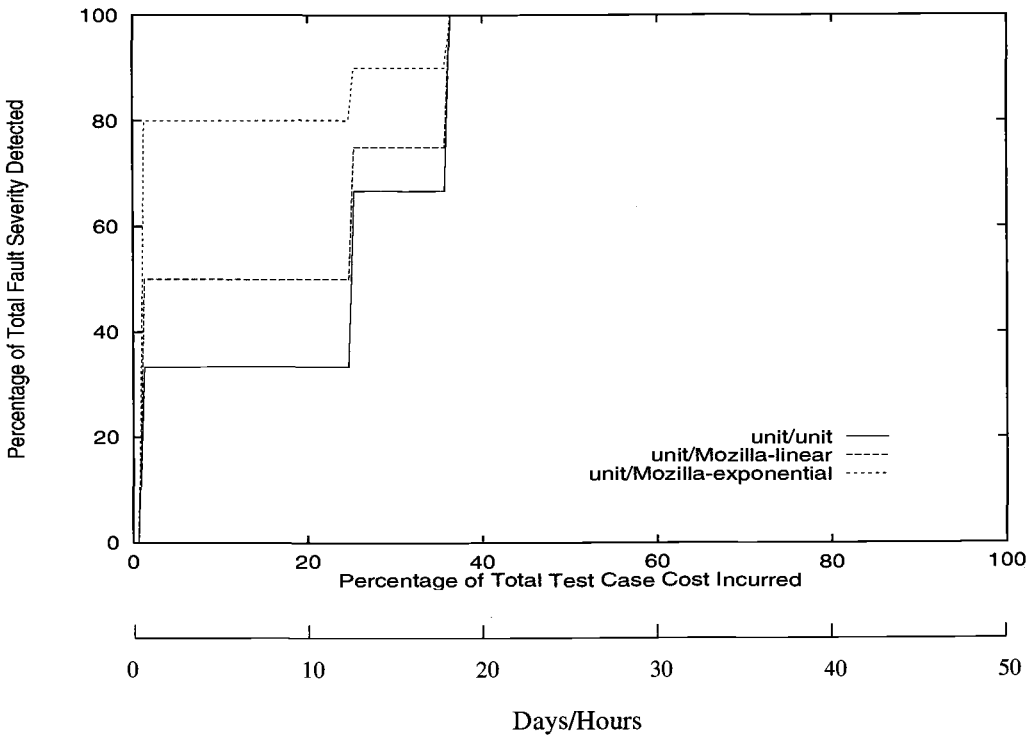


FIGURE 3.13: $APFD_C$ curves for Practical Question 2.

executed. In this case, under the Mozilla-exponential distribution, 80% of the total fault severity has been accounted for within 4 days, whereas under the Mozilla-linear distribution the same severity is not accounted for until after approximately 19 days. These savings may be significant if the product must be shipped before the testing process is completed, because the most severe faults are more likely to have been detected.

Suppose, however, that the execution of the test suite is accomplished over the weekend and its cost can be measured in hours instead of days. In this case, the differences between the distributions may not be practically significant. Thus, as

the overall testing costs diminish, the impact of choosing a particular distribution is lessened.

Practical Question 3. What considerations should I attend to when specifying test cost and fault severity distributions?

Applying linear transformations to a scale of test costs or fault severities does not affect $APFD_C$ values. For example, given a three-level fault severity ranking scheme, choosing severity values 1, 2, and 3 is equivalent to choosing 10, 20 and 30 where evaluation of test case orders is concerned, because it does not affect the units-of-fault-severity-detected-per-unit-test-cost-values of test cases relative to one another. In contrast, applying non-linear transformations to a scale of test case costs or fault severities can affect $APFD_C$ values because it can alter the relative worth of test case orders as shown by the Mozilla-exp distribution presented above.

3.7 Conclusions

This chapter introduced our cost-cognizant metric $APFD_C$ and cost cognizant prioritization techniques. It reported on a case study to explore effects of varying factors such as scale combinations used to represent cost cognizant information, ways of incorporating this information into prioritization techniques, and ways of estimating cost cognizant information. The intention of the case study was to show how different techniques and scales affect the $APFD_C$ metric and how these techniques compare to each other depending on different scales.

Our study is not intended to decide which scale to employ, it rather shows how scales and other factors affect prioritization outcomes. Scale choice should

be decided upon analysis that involves estimation of fault severities and test costs under real industrial settings (that we do not have currently access to).

CHAPTER 4

GENERAL FRAMEWORK FOR PRIORITIZATION

4.1 Introduction and Motivation

The previous chapters have presented several prioritization techniques, and provided algorithms for expressing those techniques. Those algorithms, however, are not fully general with respect to prioritization, because there are other techniques that cannot be expressed by them. For example, Kim and Porter [103] designed a prioritization technique that cannot be represented by any of our algorithms.

In this chapter, we present a general framework for prioritization, that allows us to represent all existing techniques by specifying a set of parameters. This framework allows us to clearly see the relationship between various techniques, easily develop similar techniques, analyze existing techniques, and develop a single general algorithm that expresses the whole set of techniques, facilitating their study and usage.

4.2 Illustration

Before proceeding with technical details we provide a simple informal presentation.

Our prioritization framework consists of a *combination/condensation structure*, a set of functions that define operations on this structure, and a set of steps applied to this structure. Informally, we define the three structural levels as elements, vectors,

Group1		Group2	
V1	V2	V1	V2
e1	e1	e1	e1
e2	e2	e2	e2
e3	e3	e3	e3
e4	e4		
e5	e5		

FIGURE 4.1: An example of a combination/condensation structure with two groups and two vectors per group.

and groups. (In the next section, we define these formally.) Figure 4.1 provides an example.

Informally, each element represents a single item of information utilized by the prioritization technique. It may include function coverage, test costs, module criticalities, change information, or fault-exposing-potential information. Vectors usually represent sets of elements that are treated in the same way, such as coverage, change information, fault-exposing-potential, etc. Groups usually represent sets of vectors used to compute a single component of the final value (produced by the combination/condensation structure). They are usually used when the final value is a tuple; thus, each group corresponds to a computation of a single tuple element.

For the st-cov-nofb technique, there is only one group, and one vector that contains binary coverage data. This technique sums elements in the vector to compute award values. For the fn-fi-cov-nofb technique, a second vector is added, containing fault index information. This technique multiplies corresponding elements in

Group1		Group2	
V1	V2	V1	V2
covF1	bfiF1	covF1	fepF1
covF2	bfiF2	covF2	fepF2
covF3	bfiF3	covF3	fepF3
covF4	bfiF4	covF4	fepF4
covF5	bfiF5	covF5	fepF5

FIGURE 4.2: The combination/condensation structure for prioritization technique fn-bfi-fep-nofb.

the two vectors and sums the results to compute award values; these award values are used to order test cases. Each element in a vector contains a single piece of information (e.g., one statement's coverage) for a whole test suite. In other words, an element is a list of values (we call them sub-elements), one per test case. When computing an award value for a given test case, proper sub-elements are used.

For the fn-bfi-fep-nofb technique, which employs a binary fault index and fault-exposing-potential information, there are two vectors in each of the two groups: coverage and binary fault index vectors in the first group, and coverage and fault-exposing-potential vectors in the second group. In each group, corresponding elements of that group's two vectors are multiplied and summed. Then, for each test case, an award value is created as a tuple consisting of two values, one from each group; this is used to order test cases. Award values, being tuples, are compared element-wise: first elements are used for sorting, and, in a case of a tie, second elements are compared. The combination/condensation structure for the award value computation for this technique is presented in Figure 4.2 where the **element combination** function is multiplication, the **condensation** function is summation, and the

group combination function is tuple creation. $CovFi$ is the binary function coverage information for function Fi, $bfiFi$ is the binary fault index for function Fi, and $fepFi$ is the fault exposing potential for function Fi. Each of $covFi$, $bfiFi$, and $fepFi$ is a vector of size $|T|$ whose components correspond to test cases from test suite T.

As we can see, the foregoing techniques fit easily into the framework. Later, we will show how the framework accommodates other techniques.

4.3 Combination/Condensation Structure

We now formally define the combination/condensation (CC) structure on which our framework is based.

We first define an *element* e. An element represents a single piece of data used by a prioritization technique. For example, an element $e \in \mathbb{E}$ can represent coverage information for a given statement s, modification information (number of lines changed) for function f, or the fault-exposing-potential for location l. A single element, however, represents this information for the *whole test suite*; thus, it contains such data for every test from the test suite. We also define the set $\mathbb{E}$ as the set of all elements used in the combination/condensation structure.

We define *sub-element* e^t to be a constituent part of element e corresponding to a given test t. We represent an *element* $e \in \mathbb{E}$ as a tuple $< e^1, e^2, ..., e^{|T|} >$ of size $|T|$, where T is a test suite. For example, if element e represents coverage information for statement s, each sub-element e^t represents coverage information for statement s with respect to test case t.

A *vector* v in our structure is a one dimensional array of elements. More formally, $v =< c_1, c_2, ..., c_{|v|} >$, where $\forall c_l \ 1 \leq l \leq |v| \quad \exists e \in \mathbb{E} \quad c_l \equiv e$. We define a set of elements that comprise a vector v to be E_v.

We define a *group* G to be a tuple of vectors, $G =< v_1, v_2, ..., v_{|G|} >$.

We define V to be a set of all vectors across all groups. $\bigcup_{v \in V} E_v \subseteq \mathbb{E}$, but this subset may be proper: some elements may not belong to any vector. We define such elements as *free* (e is free iff $\forall v \in V \ e \notin E_v$).

All vectors that belong to the same group must be *compatible*, defined as having the same number of elements. Vectors across different groups need not be compatible.

For each group G, we define $M_G = (m_{i,j})$ to be the matrix whose columns are the vectors in G, so $m_{i,j}$ represents the i-th component of the j-th vector in G.

Finally, at the uppermost level, we define a *super-group* SG as a set of groups.

Next, we define several functions that operate on this CC structure.

First, for each group G containing $|G|$ vectors, we define an *element combination function*

$$f^G_{\text{element\_combine}}(x_1, x_2, ..., x_{|G|}, \alpha) \tag{4.1}$$

This function takes a slice $M^G_{i,1...|G|}$ (an array $< x_1, x_2, ..., x_{|G|} >$ where x_l is the i-th component of the l-th vector in G). This function also has argument α which will be explained later. Each group has its own function $f^G_{\text{element\_combine}}$.

Next, for each group G, we define a *condensation function*

$$f^G_{\text{condensation}}(y_1, y_2, ..., y_k, \alpha), \tag{4.2}$$

where k is the number of elements in each vector in group G. This function takes the array $< y_1, y_2, ..., y_k >$, where each y_i is the result of applying an *el-*

ement combine function to the i-th slice of matrix M_G (defined earlier) ($y_i = f^G_{\text{element\_combine}}(m_{i,1}, m_{i,2}, ..., m_{i,|G|}, \alpha) \quad \forall i, \quad 1 \leq i \leq k$), and an argument α (explained later). Each group has its own *condensation function.*

Finally, we define a *super-group combination function*

$$f_{\text{group\_combine}}(z_1, z_2, ..., z_{|SG|}, \alpha) \tag{4.3}$$

This function takes an array $< z_1, z_2, ..., z_{|SG|} >$ (z_j corresponds to group $G_j \in SG$) and an argument α (explained later). Each z_j is produced by a *condensation function*: $z_j = f^{G_j}_{\text{condensation}}(y_1, y_2, ..., y_k, \alpha)$.

We call the *element combination function*, *condensation function*, and *super-group combination function*; *structure functions* (SF).

We call this sub-element/element/vector/group/supergroup structure, together with the structure functions, a *combination/condensation structure.*

Let CCF be a particular combination/condensation structure. We define a function $F_{CC}(CCF, \mathbb{E}, \alpha)$ which takes CCF, set $\mathbb{E}$ of elements, and an argument α (explained later), and produces the result of applying the structure functions to the elements of $\mathbb{E}$.

On first glance, a combination/condensation structure might seem to have rather limited power to express functions. However, in the most general case, combined with an appropriate condensation function, a combination/condensation structure has sufficient power to express any function on a given set of elements $\mathbb{E}$. For example, to express function $F(x_1, ..., x_n)$, we can create the structure consisting of a single group and a single vector with the vector containing $x_1, ..., x_n$. We set element combination and group combination functions to $g(y) = y$, and use function $F(...)$ in the place of the condensation function. Thus, our structure with

its structure functions will represent the given function $F(...)$. Our prioritization framework, described in the next section, uses this structure to express prioritization techniques. It will be shown later that the structure functions used in the combination/condensation structure employed by the prioritization framework are trivial (involve only simple algebraic expressions) for all considered prioritization techniques.

4.4 Framework

The prioritization algorithms described in this thesis and the literature come in two flavors: **without feedback** and **with feedback**. Given a test suite T, algorithms **without feedback** consist of two steps: (1) compute test award values and (2) sort tests based on those award values. On the other hand, algorithms **with feedback** use an iterative approach, recalculating award values after each test is prioritized. Algorithms **without feedback**, then, can be seen as a special case of algorithms **with feedback**, where data structures are not updated during each iteration.

In the framework that we now present, we use an iterative approach in which we apply combination/condensation structures to compute award values and to alter elements each time a new test is selected. The main idea is to modify elements $e \in E$, given a newly selected test[1].

[1] Because techniques without feedback are a special case of techniques with feedback, this framework will describe them too.

This framework uses two CC structures: one, CCF_{award}, for award value computation and another, CCF_{update}, for updating elements from $\mathbb{E}$, where $\mathbb{E}$ is the set of all elements used in the framework.

To compute award values, for each test $t \in T$, we obtain $a_t = F_{CC}(CCF_{\text{award}}, E, t)$.

To select the test with the best award value, we compute $t_s = f_{\text{best}}(\vec{a})$. Function f_{best} takes a vector of award values $\vec{a}$ and finds the test (id) with the best award value.

After a new test is selected, the framework algorithm updates all elements $e \in \mathbb{E}$. To compute a new value for an element e, several steps are taken. First, for each test $t_u \in T$ and for each element $e \in \mathbb{E}$, we compute $u_e^{t_u} = F_{CC}(CCF_{\text{update}}, E, < t_s, t_u, e >)$, using CC structure CCF_{update}, where t_s is the test selected in the previous step. Second, we update every element $e \in \mathbb{E}$ for every test $t_u \in T$ $val(e^{t_u}) = f_{\text{update}}(e^{t_u}, u_e^{t_u})$. We define $val(x)$ to be the value of x.

The framework is applied as follows: determine the set of elements E in all structures, determine the CC structure CCF_{award} for award computation containing a subset of E, determine CC structure CCF_{update} for update computation containing a subset of E, determine initial values for all elements in $\mathbb{E}$, determine a function for element updating f_{update}, decide on a sorting function f_{best}, and finally, apply the framework algorithm that initializes all elements and computes award values, selects a test, and updates elements until the halting condition is satisfied.

The framework algorithm that is used to implement prioritization techniques is presented as Algorithm 3. Lines 3-7 compute initial values for every element. Lines 8-12 set element values to the values computed in lines 3-7. Lines 13-32 implement the main loop which computes the prioritized test case sequence. Lines 14-16 compute test award values. Line 17 finds the test with the highest award

Algorithm 3 The prioritization framework algorithm.

```
1:  Initialize elements in 𝔼
2:  List = ϵ
3:  for all t_u ∈ T do
4:    for all e ∈ 𝔼 do
5:      x_e^{t_u} ← f_update(e^{t_u}, F_CC(CCF_update, E, < nil, t_u, e >))
6:    end for
7:  end for
8:  for all t_u ∈ T do
9:    for all e ∈ 𝔼 do
10:     val(e^{t_u}) ← x_e^{t_u}
11:   end for
12: end for
13: loop
14:   for all t ∈ T do
15:     a_t ← F_CC(CCF_award, E, t)
16:   end for
17:   t_s ← f_best(a⃗)
18:   if a_{t_s} = nil then
19:     HALT
20:   end if
21:   Add t_s into List
22:   for all t_u ∈ T do
23:     for all e ∈ 𝔼 do
24:       x_e^{t_u} ← f_update(e^{t_u}, F_CC(CCF_update, E, < t_s, t_u, e >))
25:     end for
26:   end for
27:   for all t_u ∈ T do
28:     for all e ∈ 𝔼 do
29:       val(e^{t_u}) ← x_e^{t_u}
30:     end for
31:   end for
32: end loop
```

value. Lines 18-20 test the halting condition. Line 19 adds the selected test to the ordered sequence. Lines 22-26 compute the new values of elements. Finally, lines 27-31 update the value of every element using values computed in lines 18-20. We define the nil value to be the lowest award value a test case can have. During comparisons, a test case with award value nil can be chosen only if there are no tests in T with award values not equal to nil.

4.5 Time Complexity

The time complexity of a given prioritization technique can vary with implementation. Usage of the general framework to implement a given technique can actually

increase its time complexity (e.g., not all elements need to be updated at every iteration). However, in this section, we will present the time complexity of the framework algorithm.

The computation time of the function defined on the CCF structure is

$$time(CCF) = time(SG) = \sum_{G \in SG} time(G) + time_{f_{\text{group\_combine}}}(|SG|) = \sum_{G \in SG} \left(time_{f_{\text{condense}}}(|v_G|) + |v_G| \times time_{f_{\text{element\_combine}}}(|G|) \right) + time_{f_{\text{group\_combine}}}(|SG|) \tag{4.4}$$

In Formula 4.4 we use the following notations:

- $time(CCF)$ is the time required to apply structure functions in combination/condensation structure CCF,
- $time(SG)$ is the time required to apply structure functions in super-group SG,
- $time(G)$ is the time required to apply structure functions in group G,
- $time_{f_{\text{group\_combine}}}(|SG|)$ is the time required to compute the **super-group combination function**,
- $time_{f_{\text{condense}}}(|v_G|)$ is the time required to compute the **condensation function** in a group G,
- $time_{f_{\text{element\_combine}}}(|G|)$ is the time required to compute the **element combination function**,

- $|SG|$ is the number of groups in the super-group,
- $|G|$ is the group size (number of vectors),
- $|v_G|$ is the vector size in group G.

The computation time for the framework algorithm is

$$\begin{aligned} time(\text{framework}) =& (|T|-1) \times (|T_{\text{selected}}|+1) \times time_{f_{\text{comparison}}} + \\ & |T| \times (|T_{\text{selected}}|+1) \times time(CCF_{\text{award}}) + \\ & |E| \times |T| \times (|T_{\text{selected}}|+1) \times time(CCF_{\text{update}}) + \\ & |E| \times |T| \times (|T_{\text{selected}}|+1) \times time_{f_{\text{update}}}, \end{aligned} \tag{4.5}$$

where T is the initial test suite, T_{selected} is the test suite that is generated by the framework, $\mathbb{E}$ is the set of all elements (defined previously), $time_{f_{\text{comparison}}}$ is the time required for one comparison, $time(CCF_{\text{award}})$ is the time required to apply structure functions to compute an award value for a given test, $time(CCF_{\text{update}})$ is the time required to apply structure functions to update a given sub-element, and $time_{f_{\text{update}}}$ is the time required to apply the f_{update} function.

As we can see, the time complexity for Algorithm 3 is polynomial relative to the input and output size if all structure functions are computable in polynomial time. One of the requirements for being able to trivially fit techniques into our framework is that all functions be polynomially computable relative to the input and output size. Intractable techniques cannot be trivially fit into our framework.

4.6 Fitting Existing Prioritization Techniques into the Framework

We now show how existing prioritization techniques can be fitted to this framework. We do this, for each technique, by defining the elements and functions that express it. We present this, for each technique, in tabular form.

In these tables, let t_{update} be the number of a test case under update, let t_{selected} be the number of a test case that has been just selected, and let t_{current} be the number of a test case for which an award value is being computed. Award values are sorted in nonincreasing order with the nil value being the lowest. Sorting uses the first tuple element as the primary key, the second tuple element as the secondary key, and so on.

Elements $e \in \mathbb{E}$ represent element entities and not just their values. However, to make expressions shorter, we will omit writing $val(...)$ for every element. Thus, in every formula, whenever needed, an element's value is implicitly referenced.

The framework performs an update to initialize elements. In the following tables, we will ignore this fact. To consider fitting the techniques more formally, we would need to slightly modify functions so that, if $t_{\text{selected}} = nil$, the original value of an element is not altered.

4.6.1 *Fault-exposing-potential Prioritization Techniques*

In the following two tables, we show how to fit techniques that utilize fault-exposing potential (called fep). These techniques are described in Section 1.2.6.

CCF_{award} and CCF_{update}	
Group 1	
Elements of vector 1	cov (coverage information)
Elements of vector 2	fep (fault-exposing potential)
Elements of vector 3	$confidence$ (confidence vector, initialized to zeroes)
Group 2	
Elements of vector 1	$unused$ (1 indicates if a test case has not been selected yet, 0 - otherwise)
Group 3	
Elements of vector 1	cov (coverage information)

CCF_{award}
Group 1
$f_{\text{element combine}}(...) = cov_i^{t_{\text{current}}} \times (1 - confidence_i^{t_{\text{current}}}) \times fep_i^{t_{\text{current}}}$
$f_{\text{condensation}}(...) =$ summation
Group 2
$f_{\text{element combine}}(...) = unused^{t_{\text{current}}}$
$f_{\text{condensation}}(x, \alpha) = x$
Group 3
$f_{\text{element combine}}(...) = 0$
$f_{\text{condensation}}(...) = 0$
$f_{\text{group combine}}(...) = \begin{cases} g_1 & \text{if } g_2 = 1 \\ nil & \text{otherwise} \end{cases}$

CCF_{update}

Group 1

$$f_{\text{element combine}}(\ldots) = \begin{cases} \langle 1-(1-confidence_i^{t_{\text{update}}}) \times fep_i^{t_{\text{selected}}}, & \text{if } confidence_i \text{ is} \\ \quad confidence_i^{t_{\text{update}}} \rangle & \text{under update} \\ \\ \langle 0,0 \rangle & \text{otherwise} \end{cases}$$

$f_{\text{condensation}}(\langle x_1, y_1 \rangle, \ldots, \langle x_n, y_n \rangle, \alpha) = \langle \sum_i x_i, \sum_i y_i \rangle$

Group 2

$$f_{\text{element combine}}(\ldots) = \begin{cases} unused^{t_{\text{update}}} & \text{if } t_{\text{selected}} \neq t_{\text{update}} \\ 0 & \text{otherwise} \end{cases}$$

$f_{\text{condensation}}(x, \alpha) = x$

Group 3

$$f_{\text{element combine}}(\ldots) = \begin{cases} cov_i^{t_{\text{selected}}} & \text{if } confidence_i \text{ is under update} \\ 0 & \text{otherwise} \end{cases}$$

$f_{\text{condensation}}(\ldots) = \text{summation}$

$$f_{\text{group combine}}(\ldots) = \begin{cases} g_1[1] & \text{if element under update is } confidence \text{ and } g_3 \neq 0 \\ g_1[2] & \text{if element under update is } confidence \text{ and } g_3 = 0 \\ 0 & \text{otherwise} \end{cases}$$

Techniques without feedback

$$f_{\text{update}}(old, new) = \begin{cases} new & \text{if the element under update is } unused \\ old & \text{otherwise} \end{cases}$$

Techniques with feedback

$$f_{\text{update}}(old, new) = \begin{cases} new & \text{if the element under update is } confidence \text{ or } unused \\ old & \text{otherwise} \end{cases}$$

4.6.2 *Change and Fault-exposing-potential Combination Prioritization Techniques*

In the following three tables, we show how to fit techniques that utilize fault-exposing potential (called fep) in combination with change information (called fi). These techniques are presented in Section 2.3.

CCF_{award} and CCF_{update}	
Group 1	
Elements of vector 1	cov (coverage information)
Elements of vector 2	fi (current change information [updated])
Elements of vector 3	fep (fault-exposing potential)
Elements of vector 4	$unused$ (1 indicates if a test has not been selected yet, 0 - otherwise)
Group 2	
Elements of vector 1	cov (coverage information)
Elements of vector 2	fi (current change information [updated])
Elements of vector 3	$unused$ (1 indicates if a test has not been selected yet, 0 - otherwise)
Group 3	
Elements of vector 1	cov (coverage information)
Elements of vector 2	fep (fault-exposing potential)
Elements of vector 3	$unused$ (1 indicates if a test has not been selected yet, 0 - otherwise)
Group 4	
Elements of vector	fep (fault-exposing potential)
Group 5	
Elements of vector	$_{\text{orig}}fi$ (original change information [not updated]
Group 6	
Elements of vector	$unused$ (1 indicates if a test has not been selected yet, 0 - otherwise)
Group 7	
Elements of vector	cov (coverage information)

CCF_{award}
All groups
$f_{\text{element combine}}(x_1, ..., x_n, \alpha) = \prod_{i=1}^{n} x_i^{t_{\text{current}}}$
$f_{\text{condensation}}(y_1, ..., y_m, \alpha) = \sum_{i=1}^{m} y_i$
$f_{\text{group combine}}(g_1, g_2, g_3, g_4, g_5, g_6, g_7, \alpha) =$ $\begin{cases} g_1 & \text{if } g_6 = 1 \text{ and technique is FI * FEP} \\ \langle g_2, g_3 \rangle & \text{if } g_6 = 1 \text{ and technique is FI and FEP double sort} \\ nil & \text{if } g_6 = 0 \end{cases}$

CCF_{update}
Group 1
$f_{\text{element combine}}(...) = \sum_t cov_i^t \times fi_i^t \times fep_i^t \times unused_i^t \times f(t, t_{\text{selected}})$, where $f(x,y) = \begin{cases} 1 & \text{if } x \neq y \\ 0 & \text{otherwise} \end{cases}$
$f_{\text{condensation}}(x_1, ..., x_n, \alpha) = \sum_i x_i$
Group 2
$f_{\text{element combine}}(...) = \langle \sum_t cov_i^t \times fi_i^t \times unused_i^t \times f(t, t_{\text{selected}}), z \rangle$, where $f(x,y) = \begin{cases} 1 & \text{if } x \neq y \\ 0 & \text{otherwise} \end{cases}$ and $z = \begin{cases} fi_i^{t_{\text{update}}} & \text{if } fi_i \text{ is the element under update} \\ 0 & \text{otherwise} \end{cases}$
$f_{\text{condensation}}(\langle x_1, y_1 \rangle, ..., \langle x_n, y_n \rangle, \alpha) = \langle \sum_i x_i, \sum_i y_i \rangle$
Group 3
$f_{\text{element combine}}(...) = \sum_t cov_i^t \times fep_i^t \times unused_i^t \times f(t, t_{\text{selected}})$, where $f(x,y) = \begin{cases} 1 & \text{if } x \neq y \\ 0 & \text{otherwise} \end{cases}$ and
$f_{\text{condensation}}(...) =$ summation

Group 4

$f_{\text{element combine}}(\ldots) = 0$

$f_{\text{condensation}}(\ldots) = 0$

Group 5

$$f_{\text{element combine}}(\ldots) = \begin{cases} {}_{\text{orig}}fi_i^{t_{\text{update}}} & \text{if } fi_i \text{ is the element under update} \\ 0 & \text{otherwise} \end{cases}$$

$f_{\text{condensation}}(\ldots) = \text{summation}$

Group 6

$$f_{\text{element combine}}(\ldots) = \begin{cases} 0 & \text{if } t_{\text{update}} = t_{\text{selected}} \\ unused^{t_{\text{update}}} & \text{otherwise} \end{cases}$$

$f_{\text{condensation}}(x, \alpha) = x$

Group 7

$$f_{\text{element combine}}(\ldots) = \begin{cases} \langle cov_i^{t_{\text{update}}}, cov_i^{t_{\text{selected}}} \rangle & \text{if } fi_i \text{ is element under update} \\ \langle 0, 0 \rangle & \text{otherwise} \end{cases}$$

$f_{\text{condensation}}(\langle x_1, y_1 \rangle, \ldots, \langle x_m, y_m \rangle, \alpha) = \langle \sum_i x_i, \sum_i y_i \rangle$

$f_{\text{group combine}}(\ldots) =$

$$\begin{cases} g_5 & \text{if element under update is } fi \text{ and awardsum} = 0 \\ g_2[2] \times (1 - g_7[2]) & \text{if element under update if } fi \text{ and awardsum} \neq 0 \\ g_6 & \text{if element under update is } unused \\ 0 & \text{otherwise} \end{cases}, \text{ where}$$

$$\text{awardsum} = \begin{cases} g_1 & \text{if FI and FEP multiplication technique} \\ g_2[1] + g_3[1] & \text{if FI and FEP double sort technique} \\ 0 & \text{otherwise} \end{cases}$$

Techniques without feedback

$$f_{\text{update}}(old, new) = \begin{cases} new & \text{if the element under update is } unused \\ old & \text{otherwise} \end{cases}$$

Techniques with feedback

$$f_{\text{update}}(old, new) = \begin{cases} new & \text{if the element under update is } fi \text{ or } unused \\ old & \text{otherwise} \end{cases}$$

4.6.3 *Coverage and Change-based Techniques*

In the following three tables, we show how to fit coverage and modification-based (change-based) prioritization techniques that are described in Section 2.3.

The final tie resolution for these techniques is coverage without feedback (the number of covered locations); however, if we need to represent techniques which do not use this, we can set $\beta = 0$. Otherwise, set $\beta = 1$.

CCF_{award} and CCF_{update}	
Group 1	
Elements of vector 1	cov (current coverage [updated])
Elements of vector 2	fi (change/modification information)
Elements of vector 3	$mcrit$ (module criticality)
Elements of vector 4	$unused$ (1 indicates if a test has not been selected yet, 0 - otherwise)
Group 2	
Elements of vector 1	$tcrit$ (test criticality)
Elements of vector 2	$tcost$ (test cost)
Group 3	
Elements of vector 1	${}_{orig}cov$ (coverage [not updated])
Group 4	
Elements of vector 1	$unused$ (1 indicates if a test has not been selected yet, 0 - otherwise)
Group 5	
Elements of vector 1	cov (coverage [updated])
Elements of vector 2	fi (change/modification information)
Elements of vector 3	$mcrit$ (module criticality)
Elements of vector 4	$tcrit$ (test criticality)
Elements of vector 5	$tcost$ (test cost)
Elements of vector 6	$unused$ (1 indicates if a test has not been selected yet, 0 - otherwise)
Group 6	
Elements of vector 1	cov (current coverage [updated])

CCF_{award}
All groups
$f_{\text{element combine}}(x_1, x_2, ..., x_n, \alpha) = \begin{cases} \frac{x_1}{x_2} & \text{if group is 2} \\ \prod_i x_i^{t_{\text{current}}} & \text{otherwise} \end{cases}$
$f_{\text{condensation}}(y_1, y_2, ..., y_m, \alpha) = \begin{cases} \beta \times \sum_i y_i & \text{for group 3} \\ \sum_i y_i & \text{otherwise} \end{cases}$
$f_{\text{group combine}} =$ $\begin{cases} \langle g_1 \times g_2, g_1, g_3 \rangle & \text{if multiplication combination technique and } g_4 = 1 \\ \langle g_1, g_2, g_3 \rangle & \text{if award value technique and } g_4 = 1 \\ \langle g_2, g_1, g_3 \rangle & \text{if ratio first technique and } g_4 = 1 \\ nil & g_4 = 0 \end{cases}$

CCF_{update}
Group 1
$f_{\text{element combine}}(...) = \sum_t (cov_i^t \times fi_i^t \times mcrit_i^t \times unused_i^t \times f(t_{\text{selected}}, t))$ where $f(x,y) = \begin{cases} 1 & \text{if } x \neq y \\ 0 & \text{otherwise} \end{cases}$
$f_{\text{condensation}}(x_1, x_2, ..., x_n, \alpha) = \sum_i x_i$
Group 2
$f_{\text{element combine}}(...) = nil$
$f_{\text{condensation}}(x_1, x_2, ..., x_n, \alpha) = nil$
Group 3
$f_{\text{element combine}}(...) = \begin{cases} \langle {}_{\text{orig}}cov_i^{t_{\text{update}}}, {}_{\text{orig}}cov_i^{t_{\text{selected}}} \rangle & \text{if } cov_i \text{ is the element under update} \\ \langle 0, 0 \rangle & \text{otherwise} \end{cases}$
$f_{\text{condensation}}(\langle x_1, y_1 \rangle, ..., \langle x_n, y_n \rangle, \alpha) = \langle \sum_i x_i, \sum_i y_i \rangle$
Group 4
$f_{\text{element combine}}(...) = \begin{cases} 0 & \text{if } t_{\text{update}} = t_{\text{selected}} \\ unused^{t_{\text{update}}} & \text{otherwise} \end{cases}$
$f_{\text{condensation}}(x, \alpha) = x$

Group 5

$$f_{\text{element combine}}(\ldots) = \sum_t (cov_i^t \times fi_i^t \times mcrit_i^t \times tcrit_i^t / tcost_i^t \times unused_i^t \times f(t_{selected}, t))$$

$$\text{where } f(x,y) = \begin{cases} 1 & \text{if } x \neq y \\ 0 & \text{otherwise} \end{cases}$$

$$f_{\text{condensation}}(y_1, y_2, \ldots, y_m, \alpha) = \sum_i y_i$$

Group 6

$$f_{\text{element combine}}(\ldots) = \begin{cases} cov_i^{t_{\text{update}}} & \text{if } cov_i \text{ is the element under update} \\ 0 & \text{otherwise} \end{cases}$$

$$f_{\text{condensation}}(y_1, y_2, \ldots, y_m, \alpha) = \sum_i y_i$$

$$f_{\text{group combine}}(\ldots) = \begin{cases} g_6 \times (1 - g_3[2]) & \text{if } \gamma \neq 0 \\ g_3[1] & \text{otherwise} \end{cases},$$

$$\text{where } \gamma = \begin{cases} g_5 & \text{if technique is multiplication} \\ g_1 & \text{otherwise} \end{cases}$$

Techniques without feedback

$$f_{\text{update}}(old, new) = \begin{cases} new & \text{if the element under update is } unused \\ old & \text{otherwise} \end{cases}$$

Techniques with feedback

$$f_{\text{update}}(old, new) = \begin{cases} new & \text{if element under update is } cov \text{ or } unused \\ old & \text{otherwise} \end{cases}$$

4.6.4 *Jones and Harrold's Technique*

In the following three tables, we show how to fit Jones and Harrold's technique, described in Section 1.2.7.1. In this technique, entities are divided into two sets (these may overlap): variable entities (updatable) and constant entities (not updatable). Also, all these entities are independently divided into two sets: contribution (to compute test contribution) and coverage (to compute test entity coverage). The coverage information is binary.

CCF_{award} and CCF_{update}	
Group 1	
Elements of vector 1	*coverage* (test entity coverage information)
Elements of vector 2	*unused* (1 indicates if a test case has not been selected yet, 0 - otherwise)
Group 2	
Elements of vector 1	*contribution* (contribution coverage information)
Elements of vector 2	*unused* (1 indicates if a test case has not been selected yet, 0 - otherwise)
Group 3	
Elements of vector 1	*flag* (it is zero initially and just after reset, meaning that test entity coverage is used; afterward it is one, meaning that test contribution is used)
Group 4	
Elements of vector 1	*unused* (1 indicates if a test case has not been selected yet, 0 - otherwise)
Group 5	
Elements of vector 1	$_{\text{orig}}elements$ (all entities including coverage and contribution ones; they are original and not updated)

CCF_{award}
Group 1
$f_{\text{element combine}}(...) = coverage_i^{t_{\text{current}}}$
$f_{\text{condensation}}(...) = \text{summation}$
Group 2
$f_{\text{element combine}}(...) = contribution_i^{t_{\text{current}}}$
$f_{\text{condensation}}(...) = \text{summation}$
Group 3
$f_{\text{element combine}}(...) = flag^{t_{\text{current}}}$
$f_{\text{condensation}}(...) = \text{summation}$
Group 4
$f_{\text{element combine}}(...) = unused^{t_{\text{current}}}$
$f_{\text{condensation}}(...) = \text{summation}$
Group 5
$f_{\text{element combine}}(...) = 0$
$f_{\text{condensation}}(...) = 0$
$f_{\text{group combine}}(...) = \begin{cases} g_1 & \text{if } g_3 = 0 \text{ and } g_4 = 1 \\ g_2 & \text{if } g_3 \neq 0 \text{ and } g_4 = 1 \\ nil & \text{if } g_4 = 0 \end{cases}$

CCF_{update}

Group 1

$$f_{\text{element combine}}(\ldots) = \begin{cases} coverage_i^{t_{\text{update}}} & \text{if it is the element under update} \\ 0 & \text{otherwise} \end{cases}$$

$f_{\text{condensation}}(\ldots) = \text{summation}$

Group 2

$f_{\text{element combine}}(\ldots) = \langle \sum_t contribution_i^t \times unused_i^t \times f(t_{\text{selected}}, t), a \rangle$, where

$$a = \begin{cases} contribution_i^{t_{\text{update}}} & \text{if it is the element under update} \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad f(x,y) = \begin{cases} 1 & \text{if } x \neq y \\ 0 & \text{otherwise} \end{cases}$$

$f_{\text{condensation}}(\langle x_1, y_1 \rangle, \ldots, \langle x_n, y_n \rangle, \alpha) = \langle \sum_i x_i, \sum_i y_i \rangle$

Group 3

$f_{\text{element combine}}(\ldots) = flag^{t_{\text{update}}}$

$f_{\text{condensation}}(\ldots) = \text{summation}$

Group 4

$$f_{\text{element combine}}(\ldots) = \begin{cases} unused^{t_{\text{update}}} & \text{if } t_{\text{update}} \neq t_{\text{selected}} \\ 0 & \text{otherwise} \end{cases}$$

$f_{\text{condensation}}(\ldots) = \text{summation}$

Group 5

$$f_{\text{element combine}}(\ldots) = \begin{cases} \langle {}_{\text{orig}}elements_i^{t_{\text{update}}}, {}_{\text{orig}}elements_i^{t_{\text{selected}}} \rangle & \text{if this element corresponds to an element from coverage or contribution, which is under update} \\ \langle 0, 0 \rangle & \text{otherwise} \end{cases}$$

$f_{\text{condensation}}(\langle x_1, y_1 \rangle, \ldots, \langle x_n, y_n \rangle, \alpha) = \langle \sum_i x_i, \sum_i y_i \rangle$

$f_{\text{group combine}}(\ldots) =$

$$\begin{cases} 0 & \text{if the element under update is } flag \text{ and } g_2[1] = 0 \\ 1 & \text{if the element under update is } flag \text{ and } g_2[1] \neq 0 \\ g_5[1] & \text{if the element under update is } entity \text{ and this element is not updatable or this element is updatable and } g_2[1] = 0 \\ g_1 \times (1 - g_5[2]) & \text{if the element under update is } entity, \text{ this element is updatable and in coverage, and } g_2[1] \neq 0 \\ g_2[2] \times (1 - g_5[2]) & \text{if the element under update is } entity, \text{ this element is updatable in contribution, and } g_2[1] \neq 0 \\ g_4 & \text{if the element under update is } unused \\ 0 & \text{otherwise} \end{cases}$$

$$f_{\text{update}}(old, new) = \begin{cases} new & \text{if the element under update is } unused, flag \text{ or } entity \\ old & \text{otherwise} \end{cases}$$

4.6.5 *Wong's Technique*

In the following two tables, we show how to fit Wong's technique, described in Section 1.2.7.1.

CCF_{award} and CCF_{update}	
Group 1	
Elements of vector 1	*coverage* (test location coverage information [updatable])
Group 2	
Elements of vector 1	$_{\text{orig}}coverage$ (original test location coverage information [not updatable])
Group 3	
Elements of vector 1	*cost* (test cost)
Group 4	
Elements of vector 1	*unused* (1 indicates if a test has not been selected yet, 0 - otherwise)

CCF_{award}
Group 1
$f_{\text{element combine}}(x, \alpha) = x^{t_{\text{current}}}$
$f_{\text{condensation}}(...) = \text{summation}$
Group 2
$f_{\text{element combine}}(x, \alpha) = 0$
$f_{\text{condensation}}(...) = 0$
Group 3
$f_{\text{element combine}}(x, \alpha) = x^{t_{\text{current}}}$
$f_{\text{condensation}}(...) = x$
Group 4
$f_{\text{element combine}}(x, \alpha) = x^{t_{\text{current}}}$
$f_{\text{condensation}}(...) = x$
$f_{\text{group combine}}(...) = \begin{cases} \frac{g_1}{g_3} & \text{if } g_4 = 1 \\ nil & \text{otherwise} \end{cases}$

CCF_{update}

Group 1

$$f_{\text{element combine}}(x, \alpha) = \begin{cases} x^{t_{\text{update}}} & \text{if this element is under update} \\ 0 & \text{otherwise} \end{cases}$$

$f_{\text{condensation}}(\ldots) = \text{summation}$

Group 2

$$f_{\text{element combine}}(x, \alpha) = \begin{cases} x^{t_{\text{selected}}} & \text{if this element is under update} \\ 0 & \text{otherwise} \end{cases}$$

$f_{\text{condensation}}(\ldots) = \text{summation}$

Group 3

$f_{\text{element combine}}(x, \alpha) = x^{t_{\text{update}}}$

$f_{\text{condensation}}(\ldots) = x$

Group 4

$f_{\text{element combine}}(x, \alpha) = x^{t_{\text{update}}}$

$f_{\text{condensation}}(\ldots) = x$

$$f_{\text{group combine}}(\ldots) = \begin{cases} g_1 \times (1 - g_2) & \text{if the element under update is } coverage \\ g_4 & \text{if the element under update is } unused \text{ and } t_{\text{update}} \neq t_{\text{selected}} \\ 0 & \text{otherwise} \end{cases}$$

$$f_{\text{update}}(old, new) = \begin{cases} new & \text{if the element under update is } coverage \text{ or } unused \\ old & \text{otherwise} \end{cases}$$

4.6.6 Srivastava and Thiagarajan's Technique

The fit for Srivastava and Thiagarajan's technique is the same as for change-based techniques, where binary change information, block coverage, uniform costs, and uniform criticalities are used.

4.7 Examples of the Framework Usage

Our framework not only allows us to express existing prioritization techniques, it also provides support for creating new ones.

Consider a prioritization technique that uses fault index information. It can be represented with one group using two vectors: the first containing fault index information and the second containing binary coverage information. The corresponding values in those vectors can be multiplied and results summed. We could add a second group with a single vector containing binary coverage information. One choice for the group combination function would be to make a tuple from the two groups' values. This tuple's first part could be used to find a test with the highest award value and its second part could be used to resolve ties. However, we could also use an arbitrary combination function such as weighted average, addition, multiplication, and so on.

Consider a technique which uses coverage, fault index, and fault-exposing-potential information. One way to structure this technique is to put all these pieces of information into three vectors, simply multiplying them and adding the results. On the other hand, we could put coverage and fault index vectors in one group and coverage and fault-exposing-potential vectors into another group. Then, the whole set of combination functions is available to choose from. Another approach is to add a third group with a single coverage vector. Just considering these examples, we can see that the framework gives us a large number of ideas for developing a great variety of prioritization techniques, given a set of program and test properties.

Several cost-cognizant techniques which utilize test criticalities and costs have been developed exclusively due to this framework. After fitting one variation of

such technique, we were able to see its other variations, inspired by choices which need to be made in order to fit the technique into the framework.

As a final example, consider Kim and Porter's technique [103], described in detail in Section 1.2.7.1. Kim and Porter describe an overall regression testing strategy in which, following a release, a regression test selection (RTS) technique is applied, and then, based on previously computed probabilities, test cases are selected one by one and executed until allotted testing time runs out. This strategy can be easily fit into the framework as shown later in this section. However, we can use the framework to actually improve the strategy. In the strategy presented by Kim and Porter, the test suite is split into three parts: test cases not selected by RTS, test cases prioritized, and test cases left out due to lack of time. Our improved technique also splits the original suite into three parts. However, it also prioritizes the three types of test cases. It prioritizes selected test cases by RTS using probabilities, until time runs out. After that, test cases omitted by RTS are prioritized. If we were to disable resource constraints (making the allotted testing time equal to the time required to run all tests), we will essentially create a prioritized list of all test cases with selected test cases first and unselected test cases second in the sequence. This ordering improves over Kim and Porter's by providing finer-grained control over test ordering.

In the following two tables, we show how to fit the technique just described.

CCF_{award} and CCF_{update}	
Group 1	
Elements of vector 1	$h_k \quad 1 \leq k \leq n$ (observations for times 1 through n)
Group 2	
Elements of vector 1	$unused$ (1 indicates if a test has not been selected yet, 0 - otherwise)
Group 3	
Elements of vector 1 ... Elements of vector m	data vector 1 needed for selection data vector m needed for selection
Group 4	
Elements of vector 1	$ttime$ (test execution time)
Elements of vector 2	$timeleft$ (time left for test execution)

CCF_{award}
Group 1
$f_{\text{element combine}}(x, \alpha) = x^{t_{\text{current}}}$
$f_{\text{condensation}}(h_1, h_2, ..., h_n, \alpha) = P_n(h_1, h_2, ..., h_n)$, where $P_k = \begin{cases} h_1 & \text{if } k = 0 \\ \gamma \times h_k + (1-\gamma) \times P_{k-1} & \text{otherwise} \end{cases}$ and γ is the smoothing constant.
Group 2
$f_{\text{element combine}}(x, \alpha) = x^{t_{\text{current}}}$
$f_{\text{condensation}}(y, \alpha) = y$
Group 3
Functions which compute award value to be used in selection: one (selected) and zero (not selected)
Group 4
$f_{\text{element combine}}(x_1, x_2, \alpha) = \langle x_1^{t_{\text{current}}}, x_2^{t_{\text{current}}} \rangle$
$f_{\text{condensation}}(y, \alpha) = y$

Original technique

$$f_{\text{group combine}}(\ldots) = \begin{cases} g_1 & \text{if } g_2 = 1 \text{ and } g_3 = 1 \text{ and } g_4[2] > 0 \\ nil & \text{otherwise} \end{cases}$$

Improved technique

$$f_{\text{group combine}}(\ldots) = \begin{cases} nil & \text{if } g_2 = 0 \\ nil & \text{if } g_3 = 1 \text{ and } g_4[2] \leq 0 \text{ and } g_2 = 1 \\ \langle g_3, g_1 \rangle & \text{otherwise} \end{cases}$$

CCF_{update}

Group 1

$f_{\text{element combine}}(\ldots) = 0$

$f_{\text{condensation}}(\ldots) = 0$

Group 2

$$f_{\text{element combine}}(\ldots) = \begin{cases} 0 & \text{if } t_{\text{update}} = t_{\text{selected}} \\ unused^{t_{\text{update}}} & \text{otherwise} \end{cases}$$

$f_{\text{condensation}}(x, \alpha) = x$

Group 3

$f_{\text{element combine}}(\ldots) = 0$

$f_{\text{condensation}}(\ldots) = 0$

Group 4

$f_{\text{element combine}}(x_1, x_2, \alpha) = x_2^{t_{\text{update}}} - x_1^{t_{\text{selected}}}$

$f_{\text{condensation}}(y, \alpha) = y$

$$f_{\text{group combine}} = \begin{cases} g_2 & \text{if the element under update is } unused \\ g_4 & \text{if the element under update is } timeleft \\ 0 & \text{otherwise} \end{cases}$$

$$f_{\text{update}}(old, new) = \begin{cases} new & \text{if the element under update if } unused \text{ or } timeleft \\ old & \text{otherwise} \end{cases}$$

CHAPTER 5

COST-BENEFIT TRADEOFFS IN PRIORITIZATION

5.1 Introduction

In many cases, prioritization techniques produce test case orderings that achieve certain objectives better than randomly ordered test cases.[1] In this dissertation, we consider improved rate of fault detection as the objective for test case orderings. Chapters 1.4 and 3 illustrated that rate of fault detection can be significantly improved by applying prioritization techniques to a test suite. A prioritized test suite that detects faults early allows engineers to address faults earlier, and this can produce savings relative to the common practice of rerunning all test cases without regard to test order.

Despite the evidence of improved rate of fault detection, the savings demonstrated in our studies do not guarantee the cost-effectiveness of prioritization techniques, because these techniques also have costs associated with them. These costs include the cost of analyzing and instrumenting the code, collecting coverage information, analyzing modifications, and executing prioritization tools.

Practitioners wishing to use prioritization techniques, and researchers wishing to study those techniques, need methods with which to assess the practical cost-effectiveness of prioritization techniques. To support such assessments, cost-

[1] Portions of this chapter have appeared previously in [121].

benefits models are required which take into account the factors affecting the costs and benefits of prioritization. Neither the APFD metric discussed in Chapter 1, nor the $APFD_C$ metric presented in Chapter 3, provide any means for assessing prioritization effectiveness in terms of such factors. In other words, these metrics do not support assessments, in terms of resources such as time or money, of the relative cost-benefits of techniques, or of whether a given technique is cost-effective in practice.

In this chapter, we present a model for assessing the relative cost-benefits of prioritization techniques. This model allows us to compare different techniques, determine when techniques would be beneficial to employ, and determine which techniques are better in given situations than others. We present the results of a case study that shows the applicability of the model in assessments.

5.2 Prioritization Cost Model

To construct our model, we consider costs associated with prioritization when a given technique is used and under common practice; we then compare those costs, and perform some simplifications of formulas.

Note that the model we present is intended to be evaluative, not predictive. As such it could be used by practitioners on historical data to investigate which techniques might have been most cost-effective for their systems and testing processes, and used to guide future efforts. It can also be used by researchers when experimenting with techniques, in order to evaluate them.

Let P be a given program, let P' be a modified version of P, let T be the test suite for P, and consider the application of prioritization relative to P and P'. Also,

let $F(T)$ be a set of regression faults in P detected by test suite T. Because we consider P to be constant, all factors depending on P will be constant relative to T. In other words, the program is treated as a constant, not as a variable.

As soon as a fault is revealed, engineers can begin working on it. If a given fault is revealed at time t_1 instead of time t_2 under a better test order, debugging is able to begin $(t_2 - t_1)$ time units earlier. The saved time could result in earlier release. Prioritization does not reduce (usually) test execution time, number of faults, or fault correction time; rather, it reduces the waiting time for faults to reveal themselves. Ideally, engineers begin fixing a fault as soon as it is revealed. Under this idealistic situation, the measure of time lost in waiting for faults to be revealed is a measure of prioritization effectiveness. We can measure prioritization benefit as the amount of reduction in wasted time.[2]

We define the following variables:

$Ca(T)$ is the cost of analysis;

$Cp(T)$ is the cost of applying the prioritization algorithm.

In discussing these variables, we distinguish two phases of regression testing — the preliminary and critical phases — these being the times before and after the release is available for testing. Preliminary phase activities may be assigned different costs than critical phase activities, since the latter may have greater ramifications for things like release time.

---

[2] There can be, of course, other metrics. One such metric measures the amount of time spent waiting until the last fault is revealed. However, this metric is too sensitive to a single fault and a single test case. Also, if each of two test orders reveals the last fault with its last test case, their values would be the same despite the fact that one test order can reveal all other faults earlier than another.

$Ca(T)$ includes the cost of source code analysis, analysis of changes between old and new versions, and collection of execution traces. $Cp(T)$ is the actual cost of running a prioritization tool, and, depending on the prioritization algorithm used, can be performed during either the preliminary or critical phase.

Given an order O of test suite T, suppose a fault i is revealed by the kth test case occurring after $d_i^O = \sum_{j=1}^{k} e_j^O$ time units, where e_j^O is the execution and validation time of test case j in the test suite under order O. Suppose that we have two orders O' and O'' of the same test suite T. Suppose fault i is revealed at time $d_i^{O'}$ if T is under order O' and the same fault is revealed at time $d_i^{O''}$ if T is under order O''. Suppose that $d_i^{O'} < d_i^{O''}$ (fault i is revealed earlier under order O'). If we have just this one fault, ordering O' is potentially beneficial relative to ordering O'', in that it saves $d_i^{O''} - d_i^{O'}$ units of time.

Consider a more complicated case in which test suite T of size n detects m faults. Let TF_i^O be the test case number under order O which first detects fault i. Let T contain n test cases.

Define:

$$r_{ik}^O = \begin{cases} 1 & \text{if test case } k \text{ in } T \text{ under order } O \text{ reveals fault } i \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

r_{ik}^O shows whether test case k under order O reveals fault i.

Let x_i^O be the set of indices of test cases, under order O, that reveal fault i. Formally:

$$x_i^O = \{k \mid \forall 1 \leq k \leq n \quad r_{ik}^O = 1\} \tag{5.2}$$

Given these definitions, $TF_i^O = \min(x_i^O)$ (the minimum of all elements in set x_i^O).

We define $delays^O$ to be the cumulative cost of waiting for each fault to be exposed while executing T under order O:

$$delays^O = \sum_{i=1}^{m} \left(\left(\sum_{k=1}^{TF_i^O} e_k^O \right) \times f_i \right) \tag{5.3}$$

In Equation (5.3), m is the number of faults, e_k^O is the run-time and validation time of test case k in T under order O, and f_i is the cost of waiting a time unit for a fault i to be exposed (e.g., paying a programmer to wait for a given fault to be exposed in order to attempt to correct it, where different faults require different programmers with different salaries). Essentially, when $\forall i \ f_i = 1$, $delays^O$ sums, for each fault, the time between the start of test suite execution and the time when this fault is first revealed.

Given that the cost savings due to the application of a prioritization technique that creates order O'' relative to using a random order[3] O' are

$$delays^{O'} - delays^{O''} - Ca(T) - Cp(T), \tag{5.4}$$

we can define the cost of random ordering O' as

$$C^{O'} = delays^{O'} \tag{5.5}$$

and the cost of prioritized order O'' as

$$C^{O''} = delays^{O''} + Ca(T) + Cp(T) \tag{5.6}$$

[3] Because there are many random orderings, this cost can be defined as the average cost over all possible random orderings of test suite T. However, because it is not practical to calculate this cost in this manner, in practice, we estimate this cost by computing an average for a fixed number of randomly chosen orderings.

Technically, the formulas for $C^{O'}$ and $C^{O''}$ do not describe specific physical costs, because the costs associated with $delays^{O}$ may not materialize in practice - they depend on factors such as availability of personnel. However, the difference between $C^{O''}$ and $C^{O'}$ gives us an upper bound on possible cost savings that could be achieved given favorable conditions. In other words, earlier detection of faults can lead to more efficient usage of resources (human and other) and earlier release dates.

It follows that prioritization is cost-effective iff:

$$delays^{O''} + Ca(T) + Cp(T) < delays^{O'} \tag{5.7}$$

We can also make several assumptions to simplify this inequality. Assume that $Ca(T) = A + a|T|$, where A is the non-test dependent cost and a is the additional cost per test case. Prioritization cost is usually linear or quadratic in some measure of the size of the test suite, depending on the algorithm. However, in our experience, the cost of performing prioritization is small compared to other costs, and because it involves only machine time, it can be neglected.

With these assumptions, inequality (5.7) becomes:

$$delays^{O''} + A + a|T| < delays^{O'} \tag{5.8}$$

Also $A \ll a|T|$ [14, 171], so we can rewrite inequality (5.8) as follows:

$$delays^{O''} + a|T| < delays^{O'} \tag{5.9}$$

This model lets us compare prioritization techniques and answer the question whether a given prioritization technique will be cost-beneficial compared to random ordering. If we need to compare two prioritization techniques 1 and 2 which

produce orders O_1 and O_2, respectively, we compute $C_1 = Ca_1(T) + Cp_1(T) + delays^{O_1}$ and $C_2 = Ca_2(T) + Cp_2(T) + delays^{O_2}$ for techniques 1 and 2, respectively (analysis and prioritization costs can differ across techniques). The technique with the lowest C_k is the most cost-beneficial.

5.3 Case Study

The foregoing model is meant to help us better evaluate cost-benefits tradeoffs involving prioritization. A fundamental research question is whether the model succeeds in this: is it more accurate, comprehensive, or realistic than previous models? We wish to know how the new model will assess cost-benefits of prioritization techniques, and how it will rank the techniques.

To gain insight into these issues, we designed and performed an exploratory case study. In the following sections, we describe our techniques, case study materials, and case study design.

5.3.1 Prioritization Techniques

As prioritization techniques we chose random prioritization, function coverage no feedback (fn-cov-nofb), function coverage feedback (fn-cov-fb), and optimal prioritization. These techniques have been described in detail in Sections 2.3 and 3.5.

5.3.2 Experiment Subjects

For these experiments we utilized a new subject program, `bash`. `Bash` is a popular shell that provides a command line interface to multiple Unix services [156]. As

Version	*Funcs.*	*Mod'd Funcs.*	*LOCs*	*Regr. Faults*
2.0	1,494	—	48,292	0
2.01	1,537	296	49,555	9
2.01.1	1,538	44	49,666	7
2.02	1,678	296	58,090	7
2.02.1	1,678	12	58,103	3
2.03	1,703	188	59,010	9
2.04	1,890	339	63,802	5
2.05-beta1	1,942	447	65,477	6
2.05-beta2	1,949	40	65,591	7
2.05	1,950	27	65,632	5

TABLE 5.1: Bash Subject

Table 5.1 shows, the latest versions of `bash` consist of over 50,000 lines of code and almost 2000 functions.

Given the complexity and cost of preparing such a large subject for experimentation, we invested significant effort in providing a supporting infrastructure so that we can reuse the `bash` subject to answer different research questions. Part of that infrastructure includes a test suite of 1168 test cases. We created this test suite using two complementary methods. First, we evaluated and refined the test suite that accompanied `bash` release 2.0. (We used the test cases from release 2.0 because they are the only ones that work across all releases.) Second, to exercise functionality not covered by the original test suite, we created additional test cases by considering the reference documentation for `bash` [156] as an informal specification. The resulting test cases exercise an average of 64% of the functions across all the versions of the system.

A second part of the infrastructure consists of a set of seeded faults created by a fault seeding process. Since we wished to evaluate the performance of regression testing techniques with respect to detection of regression faults, 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 this study, to become familiar with `bash` and to insert regression faults into the program versions. We then determined which faults were exposed by each test case. The numbers of faults utilized in the our experiments are reported in column five of Table 5.1.

Finally, we used various tools we developed to perform program instrumentation and prioritization as described in Chapter 3.

5.3.3 Case Study Design

Our case study examines the effectiveness of our cost model for assessing cost-benefits tradeoffs of the four prioritization techniques considered. For the new model, to compare savings or costs of techniques, we use the formula $delays^{O'} - delays^{O''} - a|T|$, where O' is random order and O'' is prioritized order.

The previous $APFD_C$ measure allows us to compare techniques, but not to assess relative cost-effectiveness. This case study, therefore, considers the relative costs of prioritization techniques under the new model, varying the ratio (programmer's cost per time unit to analysis cost per test case). We assume that all test cases have uniform costs.

Our case study manipulated two independent variables: the prioritization technique and the ratio of the cost of a programmer waiting one time unit to analysis cost per test case. The dependent variables are the outputs of our model, and in-

clude costs of, or savings achieved by, the application of a particular prioritization technique.

5.3.4 Results and Analysis

Our case studies manipulate cost ratio and regression testing technique to evaluate the cost model.

Figures 5.1, 5.2, and 5.3 show the cost-benefits tradeoffs among prioritization techniques by varying the relationship between programmer's cost and analysis cost. The X axes show the `bash` version and the Y axes show the cost $delays^{O'} - delays^{O''} - an$, where O'' is the order of the test suite produced by a given prioritization technique, O' is the order produced by the random method, a is the analysis cost per test case, and n is the size of the test suite. Higher values mean higher savings due to prioritization. Different lines correspond to different values for the ratio of programmer's costs to analysis costs. We can see that when this ratio is small (1:1, 10:1), there are hardly any savings over random ordering. However, as the ratio increases to 100:1 and 1000:1, the savings become more obvious.

Where comparison between our three prioritization techniques are concerned, differences were relatively small. Optimal naturally gave the best results in terms of savings. However, optimal cannot actually be implemented if faults are not known. Fn-cov-fb was the next technique to show substantial savings in costs; for example, when the ratio was 10:1, for version 7, fn-cov-fb saved 2,714,220 cost units. Fn-cov-nofb showed slightly more modest cost savings, and on some versions, was not even beneficial relative to random ordering.

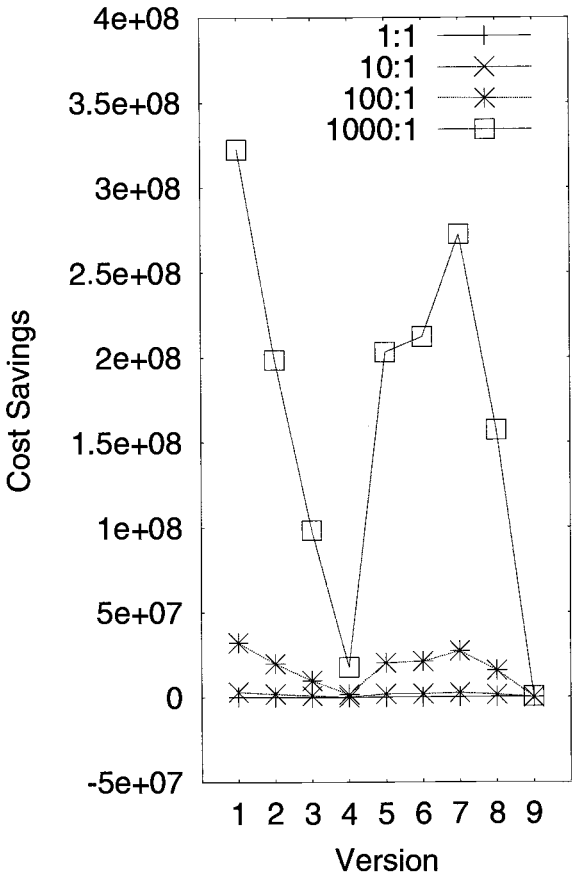


FIGURE 5.1: Differences between fn-cov-fb and random across various cost ratios.

5.4 Discussion and Conclusions

The relative ranking of the prioritization techniques we considered was not affected by varying ratios. Also, the relative prioritization technique ranking produced by the *delays* measure was similar to the ranking produced by the $APFD_C$ metric. This means that the cost model can be used, similar to the APFD metric, to assess the relative effectiveness of prioritization techniques, find the best technique, and

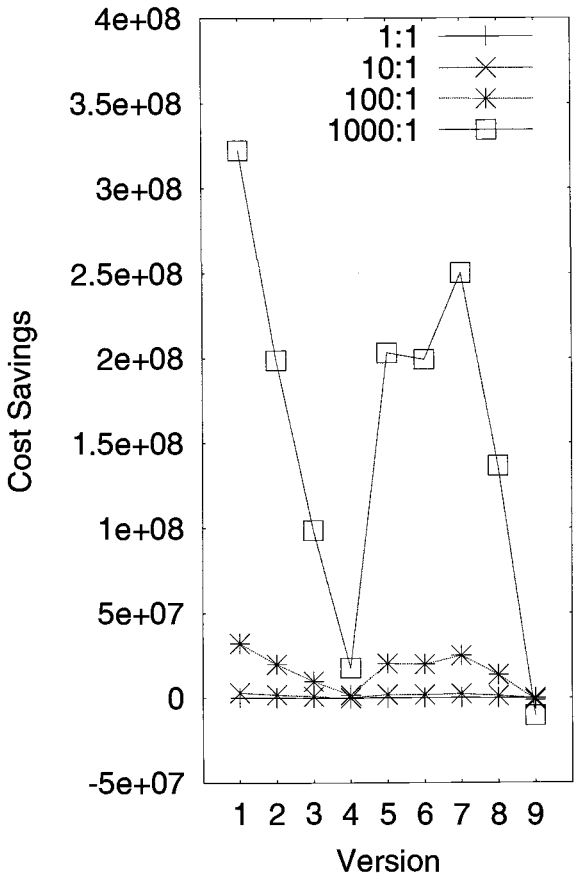


FIGURE 5.2: Differences between fn-cov-nofb and random across various cost ratios.

find the worst technique. However, the new model clearly provides a more accurate assessment of how much practical benefit can be obtained through a prioritization technique – the main consideration when incorporating a technique into a testing process. The model can thus allow us to decide when a given prioritization technique is cost-beneficial and which technique would be the most beneficial to apply, saving the greatest amount of resources.

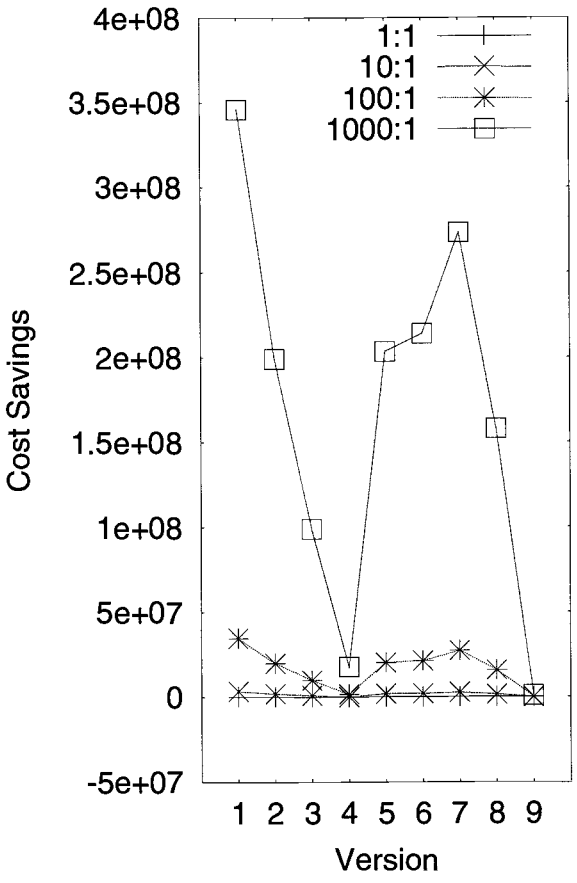


FIGURE 5.3: Differences between optimal and random across various cost ratios.

We have primarily considered assessment after the fact of regression testing; this can be used by practitioners to make decisions analyzing historical data, or by researchers to evaluate experimental results. A second subject of interest concerns the ability to predict the cost-effectiveness of prioritization techniques in advance, before testing starts. We consider this issue in Chapter 7.

CHAPTER 6

UNDERSTANDING FACTORS THAT INFLUENCE PRIORITIZATION EFFECTIVENESS

6.1 Introduction

The empirical studies reported in Chapters 1.4 and 3 have shown that prioritization techniques can be cost-effective. These studies have also shown, however, that the cost-effectiveness of prioritization techniques varies with several factors, including the characteristics of the software under the test, the attributes of the test cases used in testing, and the modifications made to create new versions of the software.

To improve prioritization effectiveness, we need to isolate these factors, determine their effect on prioritization effectiveness, and, finally, lay groundwork for improving effectiveness by suggesting guidelines for choosing prioritization strategies.

Elbaum et al. [46] identified several metrics on program structure, test suite composition, and changes that explain some of the variation in prioritization effectiveness. However, this paper did not individually investigate the factors underlying these metrics.

Engineers unaware of the relationship between change patterns and testing technique cost-effectiveness can make poor choices in defining prioritization processes. These choices may include: (1) designing regression test suites that cannot be effectively prioritized, (2) integrating changes into a build that adversely affect prior-

itization, and (3) selecting inappropriate prioritization techniques. Engineers aware of causal factors could make better choices, and perform prioritization more cost-effectively.

In this chapter, we explore two sets of factors: one involving test suites and another involving the program and modifications. First, in Section 6.2, we describe an experiment on the effects of test suite composition on the effectiveness of prioritization. Second, in Section 6.3, we describe an experiment on the effect of modifications on the effectiveness of prioritization.

6.2 Test Suite Composition Effects

6.2.1 Introduction

The effectiveness of prioritization techniques varies with characteristics of test suites (Chapter 1.4 and [172, 173]).[1] One prominent factor in this variance involves the way in which test inputs are composed into test cases within a test suite. For example:

- A test suite for a word processor might contain just a few test cases that start up the system, open a document, issue hundreds of editing commands, and close the document, or it might contain hundreds of test cases that each issue only a few commands.

[1] Portions of this section have appeared previously in [177].

- A test suite for a compiler might contain several test cases that each compile a source file containing hundreds of language constructs, or hundreds of test cases that each compile a source file containing just a few constructs.

- A test suite for a class library might contain a few test drivers that each invoke dozens of methods, or dozens of drivers that each invoke a few methods.

These examples expose important choices in test suite design, and faced with such choices, test engineers may wonder how to proceed. Textbooks and articles on testing provide various (and sometimes contradictory) forms of advice. Beizer [12, p. 51], for example, writes: "It's better to use several simple, obvious tests than to do the job with fewer, but grander, tests." Kaner et al. [97, p. 125], suggest that large test cases can save time, provided they are not overly complicated, in which case simpler test cases may be more efficient. Kit [106, p. 107] suggests that when testing valid inputs for which failures should be infrequent, large test cases are preferable. Hildebrandt [86] argues that small test cases facilitate debugging. Bach [5] states that small test cases cause fewer difficulties with cascading errors, but large test cases are better at exposing system level failures involving interactions between software components.

Most of the foregoing statements refer primarily to test case size, but this is an oversimplification: the underlying factors involved are more complex. For this work, we consider two specific factors related to test suite composition: *test suite granularity* and *test input grouping*. These characteristics both pertain to the way in which test engineers group individual test inputs into test cases within test suites. Test suite granularity pertains to the size of the test cases so grouped (the number of inputs, or amount of input applied, per test case), whereas test input grouping

pertains to the content of test cases (the degree of hetero- or homogeneity displayed by the inputs composing a test case). We define these characteristics more precisely in Section 6.2.2, and provide precise measures for these characteristics in Section 6.2.4.1.

Despite the apparent importance of test suite composition, and the contradictions among the foregoing suggestions, in our search of the research literature we find little formal examination of the cost-benefits tradeoffs associated with test suite granularity and test input grouping. A thorough investigation of these tradeoffs, and the implications they hold for prioritization, could help test engineers design test suites that better support cost-effective prioritization.

We have therefore designed and performed a controlled experiment examining the effects of test suite granularity and test input grouping on the effectiveness of test case prioritization. Our experiment considers the application of several prioritization techniques across ten releases each of two substantial software systems, using seven different levels of test suite granularity, and two different approaches for grouping test cases into suites of varying granularity. We measure and analyze the effects of granularity, technique, program, and grouping on the rate of fault detection achieved by prioritization.

Our results show that test suite granularity significantly effects the effectiveness of prioritization techniques, while test input grouping itself has limited effects.

6.2.2 *Test Suite Granularity and Test Input Grouping*

Following Binder [15], we define a *test case* to consist of a pretest state of the system under test (including its environment), a sequence of test inputs, and the expected test results. We define a *test suite* to be a set of test cases.

Definitions of test suite granularity and test input grouping are harder to come by, but the testing problem we are addressing is a practical one, so we begin by drawing on examples.

Test engineers designing test cases for a system identify various *testing requirements* for that system, such as specification items, code elements, or method sequences. Next, they must construct test cases that *exercise* these requirements. An engineer testing a word processor might specify sequences of editing commands, an engineer testing a compiler might create sample target-language programs, and an engineer testing a class library might develop drivers that invoke methods. The practical questions these engineers face include how many and which editing commands to include per sequence, how many and which constructs to include in each target-language program, and how many and which methods to invoke per driver, respectively.

It is questions such as these that we wish to answer, and the questions involve many cost-benefits tradeoffs. For example, if the cost of performing setup activities for individual test cases dominates the cost of executing those test cases, a test suite containing a few large test cases can be less expensive than a suite containing many small test cases. Large test cases might also be better than small ones at exposing failures caused by interactions among system functions. Small test cases, on the other hand, can be easier to use in debugging than large test cases, because

they reduce occurrences of cascading errors [5] and simplify fault localization [86]. Further, in test cases composed of large numbers of test inputs, inputs occuring early in the test cases may prevent test inputs that appear later in those test cases from exercising the requirements they are intended to exercise, by causing subsequent test inputs to be applied from system states that differ from those intended.

In part, the foregoing examples involve *test case size*, a term used informally in [5, 12, 97, 106] to denote notions such as the number of commands applied to, or the amount of input processed by, the program under test, for a given test case. However, there is more than just test case size involved: when engineers increase or decrease the number of requirements covered by each test case, this directly determines the number of individual test cases that must be created to cover all the requirements. Thus, as expressed by Beizer [12], the choice is not just between "large" and "small" tests, but between "several simple, obvious tests" and "fewer, but grander, tests".

The interaction of test case size and number of test cases is at least partly responsible for creating the cost-benefits tradeoffs described above. One phenomenon we wish to study, then, involves the effects that occur relative to prioritization when, in the course of designing a test suite to cover requirements, engineers compose test inputs into specific size test cases in a test suite. We use the term *test suite granularity* to describe a partition on a set of test inputs into a test suite containing test cases of a given size. (Section 6.2.4.1 presents a precise metric for this construct.)

An additional factor that may influence the effects of choices in test suite design involves the relationship between the particular test inputs that are assembled into individual test cases. For example, a typical approach in test development and automation is for test engineers to group together, into individual test cases, test

inputs that address similar functionality; this can be distinguished from approaches that group test inputs in other ways (e.g., into test cases developed per engineer, or per team). We use the term *test input grouping* to describe this factor. (Section 6.2.4.1 provides a precise metric for this construct.)

As thus described, test suite granularity concerns the sizes of individual test cases, but not their content, and test input grouping concerns the content of individual test cases, but not their size. Together these two terms can be said to represent *test suite composition*, but as we shall show, the two factors are largely orthogonal, and by treating them separately, we are able to examine both their individual and combined roles in affecting the effectiveness of test case prioritization techniques.

6.2.3 Program Subjects

We use two program subjects, `bash` and `emp-server`. The `bash` program subject was described in Section 5.3.2.

`Emp-server` is essentially a transaction manager: its main routine consists of initialization code followed by an event loop in which execution waits for receipt of a user command. `Emp-server` is invoked and left running on a host system; a user communicates with the server by executing a `client` that transmits the user's inputs as commands to `emp-server`. When `emp-server` receives a command, its event loop invokes routines that process the command, then waits to receive the next command. As `emp-server` processes commands, it may return data to the client program for display on the user's terminal, or write data to a local database (a directory of ASCII and binary files) that keeps track of game state. The event loop and program terminate when a user issues a "quit" command. Table 6.1 shows

Program	*Version*	*Functions*	*Changed Functions*	*Lines of Code*
emp-server	4.2.0	1,188	—	63,014
emp-server	4.2.1	1,188	51	63,014
emp-server	4.2.2	1,197	245	63,658
emp-server	4.2.3	1,196	157	63,937
emp-server	4.2.4	1,197	9	63,988
emp-server	4.2.5	1,197	101	64,063
emp-server	4.2.6	1,197	32	64,108
emp-server	4.2.7	1,197	156	64,439
emp-server	4.2.8	1,189	52	64,381
emp-server	4.2.9	1,189	12	64,396

TABLE 6.1: Experiment Subjects

the numbers of functions and lines of executable code in each of the ten versions of `emp-server` that we considered, and for each version after the first, the number of functions changed for that version (modified or added to the version, or deleted from the preceding version).

To examine our research questions — whether test suite granularity or test input grouping has an effect on the effectiveness of prioritization techniques — we required test cases for our subject programs. These test cases had to be structured in a realistic manner, but also in a manner that facilitates the controlled investigation of the effects of test suite granularity and test input grouping, following the methodology outlined in Section 6.2.4.1. The approaches used to create and automate these test cases, which differed between our subject programs, were as follows.

6.2.3.1 *Emp-server Test Cases and Test Automation*

No test cases were available for `emp-server`. To construct test cases we used the `Empire` information files, which describe the 196 commands recognized by `emp-server`, and the parameters and environmental effects associated with each. We treated these files as informal specifications for system functions and used them, together with the category partition method [149], to construct a suite of test cases for `emp-server` that exercise each parameter, environmental effect, and erroneous condition described in the files.

We deliberately created the smallest test cases possible, each using the minimum number of commands necessary to cover its target requirement. Each test case consists of a sequence of between one and six lines of characters (average 1.2 lines per test case), and constitutes a sequence of inputs to the `client`, which the `client` passes to `emp-server`. Because the complexity of commands, parameters, and effects varies widely across the various `Empire` commands, this process yielded between one and 38 test cases for each command, and ultimately produced 1985 test cases. These test cases constituted our test grains, as well as our test cases at granularity level G1. We then used the two sampling procedures described in Section 6.2.4.1 to create random and functional grouping test suites at granularity levels G2, G4, G8, G16, G32, and G64, the sizes of which are shown in Table 6.2.

To execute and validate test cases automatically, we created test scripts. Given test suite T, for each test case t in T these scripts: (1) initialize the Empire database to a start state; (2) invoke `emp-server`; (3) invoke a `client` and issue the sequence of inputs that constitutes the test case to the client, saving all output returned to the client for use in validation; (4) terminate the client; (5) shut down

	emp-server	bash
G1	1985	1168
G2	993	584
G4	497	292
G8	249	146
G16	125	73
G32	63	37
G64	32	19

TABLE 6.2: Test Cases per Granularity Level

emp-server; (6) save the contents of the database for use in validation; and (7) compare saved client output and database contents with those archived for the previous version. By design, this process lets us apply (in step 3) all of the test inputs contained in a test case, at all granularity levels.

As described in Section 6.2.4.1, our sampling procedure, applied to emp-server, does create non-uniform test cases, due to the particular sizes of buckets corresponding to test cases of different functionalities. Table 6.3 illustrates, for each test suite granularity, the percentage of non-uniform grouping test cases occurring at that granularity. When analyzing our results we take care to account for the effects of non-uniform groupings.

6.2.3.2 Bash Test Cases and Test Automation

The bash test suite was described in Section 5.3.2. As described there, the test suite contains 1168 test cases, exercising an average of 64% of the functions across all the versions. Each test case in the new test suite contains between one and 54

Program	G2	G4	G8	G16	G32	G64
emp-server	0.5	11.0	28.0	65.0	88.0	100.0
bash	0.5	2.0	4.5	9.6	21.6	36.8

TABLE 6.3: Percentage of Functional Test Cases with Non-Uniform Groupings.

lines. Each line constitutes an instruction consisting of `bash` or `Expect` [119] commands depending on the type of test case,[2] that can be executed on an instance of `bash`. The 1168 test cases constituted our test grains, and test cases at granularity level G1. As with `emp-server`, we then followed the procedure described in Section 6.2.4.1 to create random and functional grouping test suites at granularity levels G2, G4, G8, G16, G32, and G64, as reported in Table 6.2.

As with `emp-server`, our sampling procedure, applied to `bash`, created non-uniform test cases. Table 6.3 displays, for each test suite granularity, the percentage of such test cases per granularity. (Due to the distribution of test cases per item of functionality tested, these percentages are lower for `bash` than for `emp-server`.)

6.2.3.3 Faults

Our experiment requires that our subjects contain regression faults. For `bash`, we used the faults seeded in it, described in Section 5.3.2. For `emp-server` we applied a similar process. This gave us, in total across all program versions, 159 faults.

---

[2] `Expect` scripts were used for test cases exercising features of `bash` that required interaction.

6.2.4 Experiments

Informally, our goal is to address the research question: "how do test suite granularity and test input grouping affect the costs and benefits of prioritization?"

More formally, we seek to evaluate the following hypotheses (expressed as null hypotheses) for test case prioritization — at a 0.05 level of significance:

H1 **(test suite granularity):** Test suite granularity does not have a significant impact on the effectiveness of prioritization techniques.

H2 **(test input grouping):** Test input grouping does not have a significant impact on the effectiveness of prioritization techniques.

H3 **(technique):** Prioritization techniques do not perform significantly differently in terms of the selected effectiveness measure.

H4 **(program):** The program under test does not have a significant effect on the effectiveness of prioritization techniques.

H5 **(interactions):** Test suite granularity and test input grouping effects across prioritization techniques and programs do not significantly differ.

To test these hypotheses we designed a controlled experiment.

6.2.4.1 Variables and Measures

Our experiment manipulated four independent variables: prioritization technique, test suite granularity, test input grouping, and program.

Prioritization Technique. We selected four test case prioritization techniques: random, fn-cov-fb, fn-bdiff-cov-fb, and optimal prioritization. Random and optimal prioritization are described in Section 2.4.3. The other two techniques are described in Section 2.3.

Test Suite Granularity. To investigate the impact of test suite granularity on the effectiveness of prioritization techniques, we needed to obtain test suites of varying granularities, in a manner that controls for other factors that might affect our dependent measures.

We considered two approaches for doing this. The first approach is to obtain or construct test suites for a program, partition them into subsets according to size, and compare the results of executing these different subsets. A drawback of this approach, however, is that it will not let us determine whether a causal relationship exists between test suite granularity and prioritization effectiveness, because it does not control for other factors that might influence those measures. To see this, suppose that T can be partitioned into two subsets, T_1 and T_2, where T_1 contains test cases of size less than s, and T_2 contains test cases of size greater than or equal to s. Suppose that we compare the effect of utilizing T_1 and T_2 and find that they differ. In this case, we cannot determine whether this difference was caused by test suite granularity, or by differences in the number or type of inputs applied in T_1 and T_2. For example, it might be the case that the types of functionality exercised by the inputs in T_2 happen to include all functionality modified to create P'. In this case, differences in performance between the two subsets could occur for reasons other than test case granularity.

The second approach that we considered is to construct test suites of varying granularities by sampling a single pool or "universe" of *test grains*. A *test grain* is a smallest input that could be used as a test case (applied from a start state and producing a checkable output) for a target program. A *sampling procedure* can select test grains to create test cases of different sizes: a test case of size s consists of s test grains. Applying this sampling procedure repeatedly to a universe of n test grains, without replacement, until none remain (partitioning the universe into n/s test cases of size s, and possibly one smaller test case), yields a test suite of *granularity level* s. Repeating this procedure many times for each of several values of s gives us test suites of different granularity levels that can be compared controlling for differences in types and numbers of inputs.

We chose this second approach, and employed seven granularity levels: 1, 2, 4, 8, 16, 32 and 64, which we refer to as G1, G2, G4, G8, G16, G32 and G64, respectively.

Test Input Grouping. We considered two different approaches for grouping test inputs into test cases at varying levels of granularity: *functional grouping* and *random grouping*.

Functional grouping test cases are composed (to the extent possible) of inputs that exercise the same functionality. To create functional grouping test cases, we began by separating the test cases in the test universe U for each subject program P into "buckets", where each bucket B_k contains the test cases in U targeting functionality k in P. Given these buckets, we considered two approaches for creating functional grouping test cases of granularity s.

- From within each bucket, randomly select groups of s test grains without replacement until fewer than s test grains remain in the bucket. Then, if any test grains remain in that bucket, let them constitute one final group (of size less than s).

- From within each bucket, randomly select groups of s test grains without replacement until fewer than s test grains remain in the bucket. Do this for each bucket. For any test grains remaining in buckets, group them into a single pool, and from them, randomly select groups of size s test cases.

The difference between these two approaches lies in their handling of test cases that remain in each bucket after the maximum number of groups of size s have been selected from that bucket. The first approach, however, has a significant drawback, in that it might yield a large number of test cases of size less than s at each granularity level (potentially as many as one per bucket). The presence of such test cases might cause a confounding effect for investigating the effects of granularity: we need to control for the number and size of test cases created at each granularity level. Thus, we chose the second approach for creating functional groupings.

One consequence of this choice relates to the fact that for most buckets in the test suites for the subjects we studied, bucket size is less than 64 (our highest granularity studied); many other buckets have size less than 32, and several have size less than 16. (Section 6.2.3 presents data on our experiment subjects and test suites.) Test suites constructed from the universes for these subjects using functional groupings thus contain progressively larger percentages of *non-uniformly grouped* test cases as granularity level increases. We take this into account in our data analyses.

Our second approach for grouping test inputs, random grouping, involves, for each level of test case granularity sought, applying our sampling procedure repeatedly to a universe of n test grains and sampling randomly across the whole universe each time (without replacement). Such a grouping provides a set of test cases, at each granularity level, equivalent in size to the set of test cases obtained with the functional grouping approach, and allows us to draw conclusions on the potential influence of functional grouping on granularity effects.

Programs. For these experiments we utilized ten successive releases of each of two substantial C programs: `emp-server` and `bash`, described in Section 6.2.3.

Dependent Variables and Measures. To investigate our hypotheses we need to measure the effectiveness of the various prioritization techniques considered. To do this we used our APFD model.

6.2.5 Experiment Design and Analysis Strategy

Our experiment has four factors (one for each independent variable) with multiple levels to ensure unbiased treatment assignment. We employ a Randomized Factorial (RF) design that has 2 levels for program, 2 levels for grouping, 7 levels for granularity, and 4 levels for technique. Each design cell has nine observations, corresponding to each of the versions (after the base version) from each program under each treatment combination. These versions constitute random effects that we do not control, and we consider them samples from a population of program versions.

The choice of a factorial design was based on the power of analysis offered by its treatment combinations, which let us interpret not only the main factors but

also their interactions. The incorporation of four factors was aimed at decreasing the variability of the results by controlling more independent variables, while at the same time increasing the generalizability of the results by observing various scenarios that might be present in the real world. However, these gains came at the cost of generating high order interactions (e.g., between 4 factors) which are extremely difficult to interpret, which led us to restrict our analyses to main effects and second order interactions.

It is interesting to note that such a factorial design is often avoided in other disciplines due to the costs of obtaining subjects for all possible combination of independent variables. Since our subjects were programs, and we had automated a large part of the experiment, we were able to gather the data necessary to comply with such a design. Still, given the effort involved in preparing subject versions (ranging, approximately, from 80 to 300 hours per version) we wanted to detect meaningful effects with a minimal number of invested resources. We decided to conservatively determine sample size by duplicating the number of versions used in an earlier version of this study [176] where significance was detected for at least one of the factors.

6.2.6 Threats to Validity

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

6.2.6.1 *Internal Validity*

To test our hypotheses we had to conduct an experiment requiring a large number of processes and tools. Some of these processes involved programmers (e.g., fault seeding) and some of the tools were specifically developed for the experiment, 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 groups of programmers. Also, we validated new tools by testing them on small sample programs and test suites, refining them as we targeted the larger subjects, and cross validating them across labs.

Having only one test suite for each test input grouping type at each granularity level in each subject might be another threat to internal validity. Although multiple test suites would have been ideal, our procedure for generating coarser granularity test suites involved randomly selecting and joining test grains, which reduces the chances of bias caused by test suite composition.

Our handling of masking effects (described in Section 6.2.4.1) might constitute a further threat to internal validity; however, as noted there, evidence suggests that such effects occur infrequently among the test cases we utilized.

6.2.6.2 *External Validity*

Three issues affect the generalization of our results. The first issue is the quantity and quality of subjects. Although using only two subjects might lessen the external validity of the study, the relatively consistent results for `bash` and `emp-`

`server` suggest that the results may generalize. Regarding subject quality, there is a large population of C programs of similar size. For example, the linux RedHat 7.1 distribution includes source code for 394 applications; the average size of these applications is 22,104 non-comment lines of code, and 19% have sizes between 25 and 75 KLOC. Still, replication of these studies on other subjects could increase the confidence in our results.

The second issue involves fault representativeness. Our fault seeding process helped us control for threats to internal validity; however, faults and fault patterns may differ in practice.

The third limiting factor is test process representativeness. Although the random and functional grouping procedures we employed to obtain coarser granularity test suites are powerful in terms of control, they constitute simulations of the testing procedures used in industry, which might also impact the generalization of the results. Complementing these controlled experiments with case studies on industrial test suites and actual faults, though sacrificing internal validity, could help.

6.2.6.3 Construct Validity

The dependent measure that we have considered is not the only possible measure of the costs and benefits of prioritization techniques. Our measures ignore the human costs that can be involved in executing and managing test suites. Our measures do not consider debugging costs such as the difficulty of fault localization, which could favor small granularity test suites [86]. Our measures also ignore the analysis time required to prioritize test cases. Our previous work has shown, however, that for the techniques considered, analysis time is much smaller than test execution

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

6.2.6.4 Conclusion Validity

The number of programs and versions we considered was large enough to show significance for most of the techniques we studied, though not for all cases. Although the use of more versions would have increased the power of the experiment, the average cost of preparing each version ranged from 80 to 340 hours, limiting the cost-effectiveness of taking additional observations.

6.2.7 Data and Analysis

In this section we investigate the effects of test suite granularity and grouping on prioritization, employing descriptive and inferential statistics.

As stated above, we analyze four prioritization techniques: random prioritization as a control (random), optimal prioritization to provide an upper bound on performance (optimal), function coverage feedback (fn-cov-fb), and function binary diff coverage feedback (fn-bdiff-cov-fb).

Figure 6.1 displays four pairs of graphs, two per technique (one per test grouping), with our measure of rate of fault detection, APFD, on the y axes. Results for both programs appear similar under the optimal technique: there was a slow but consistent decrease in APFD as granularity increased, independent of grouping technique. This was expected, because having more test cases provides more opportunities for prioritization; still, the differences were small.

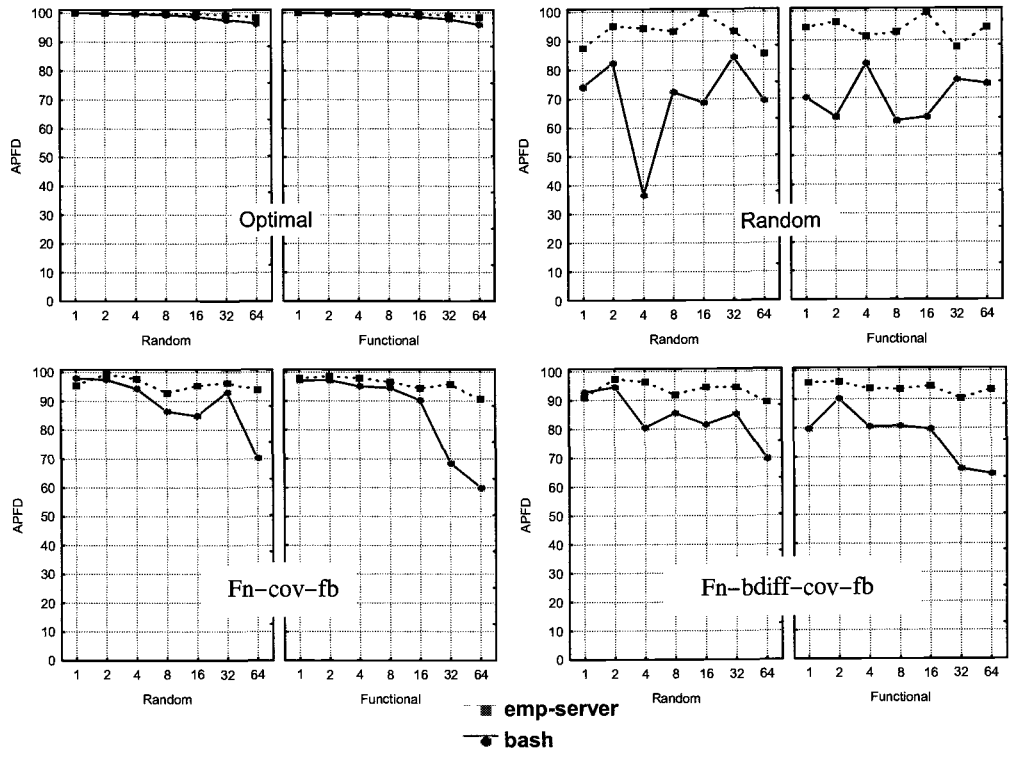


FIGURE 6.1: APFD values for test case prioritization.

Under random prioritization results varied more widely. Per definition, this variation was expected and is not directly attributable to increases in granularity or changes in functional grouping. Fn-cov-fb exhibited a decrease in APFD as granularity increased. The rate of decrease is greater for this technique than for the optimal technique, and more obvious for `bash` than for `emp-server`. Similar tendencies can be observed for fn-bdiff-cov-fb, confirming that lower granularities offer better opportunities for prioritization in general.

The Anova presented in Table 6.4 determines whether the differences observed in the graphs are significant. As the table shows, program, granularity and technique

All Techniques					
Variable: *APFD.*					
Source	SS	D.F.	MS	F	p
Program	32369	1	32369	343.40	*0.00*
Granularity	8458	6	1410	14.95	*0.00*
Grouping	242	1	242	2.57	0.11
Technique	41101	3	13700	145.35	*0.00*
*Program*Granularity*	3512	6	585	6.21	*0.00*
*Program*Grouping*	477	1	477	5.06	*0.03*
*Granularity*Grouping*	3223	6	537	5.70	*0.00*
*Program*Technique*	16637	3	5546	58.83	*0.00*
*Granularity*Technique*	8664	18	481	5.11	*0.00*
Grouping*Technique	645	3	215	2.28	0.078
Error	90395	959	94		

TABLE 6.4: Prioritization Anova

all had a significant effect on the value of APFD. This means that the two programs generated significantly different APFD averages, that increasing granularity resulted in significantly inferior APFD values, and that the chosen APFD value can change significantly based on the prioritization technique that is in place. We could not reject, however, the null hypothesis for the grouping factor for prioritization: grouping seems not to significantly affect results.

Several of the interactions were also significant, which helps us better understand how differences in one factor depend on other factors. To clarify the meanings of these interactions we constructed three plots (Figure 6.2). The topmost plot shows that both programs are affected differently by changes in granularity. The middle plot illustrates how the performance of prioritization techniques is also af-

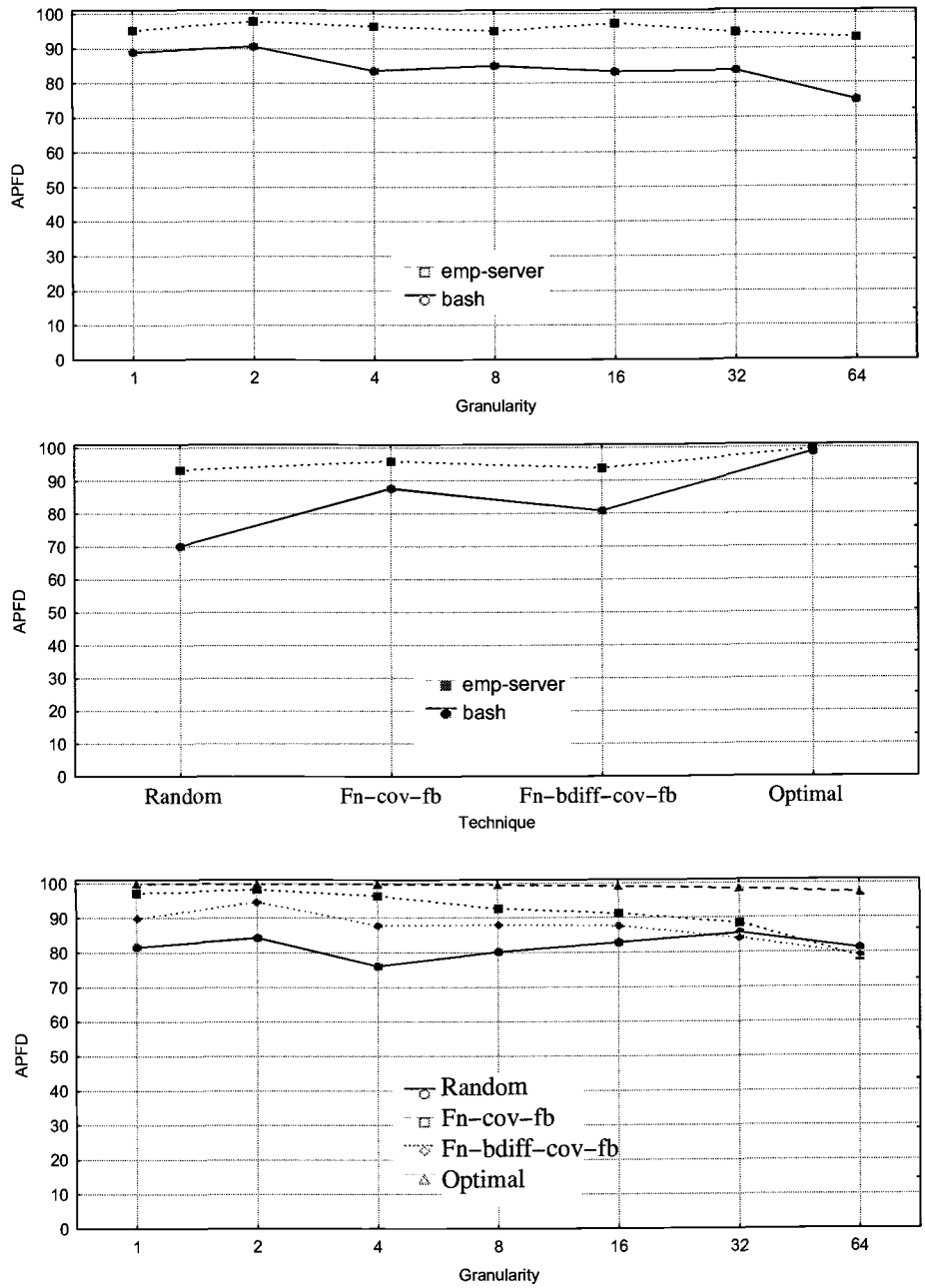


FIGURE 6.2: Prioritization interactions.

fected by attributes of the program's characteristics. Both of these graphs illustrate similar APFD tendencies for both programs, confirming the significance of the main effect. The lower graph shows how the techniques are affected differently by the variation in granularity. Except for the random technique, the rest of the techniques exhibit similar trends, with optimal exhibiting small decreases in APFD, while fn-cov-fb and fn-bdiff-cov-fb exhibit an average decrement in APFD of approximately 12% as granularity increases from level G1 to level G64. The variation in random was expected and does not affect the validity of the previous Anova interpretation.[3] Overall, these interactions do not seem to contradict or obscure the significance found in the main effects.

We would not want to conclude, simply on the basis of APFD values, that low granularity suites are necessarily better than high granularity suites, because such suites may also differ in terms of execution time. Thus, we also considered the effect of granularity on execution time. Figure 6.3 shows that test execution time decreased as granularity increased, independent of grouping strategy or program. For example, under the random grouping, test execution time for `bash` was reduced from 782 minutes at granularity level G1 to 222 minutes at granularity level G64, and for `emp-server` was reduced from 505 minutes at level G1 to 26 minutes at level G64. We formally investigated these tendencies relative to our hypotheses by performing an analysis of variance (ANOVA), that included the sources of variation considered, the sum of squares, degrees of freedom, mean squares, F value, and p-value for each source. Because we set alpha to 0.05, and the p-value represents the

[3] Since the grouping factor was not significant, we do not present interactions related to grouping.

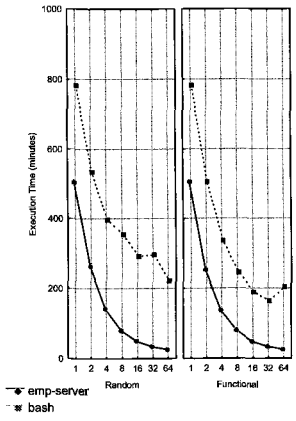


FIGURE 6.3: Test execution time for random and functional groupings across test suite granularities (x-axis), averaged across versions.

smallest level of significance that would lead to the rejection of a null hypothesis, we reject an hypothesis when p is less than alpha. The results (Table 6.5) indicate that program and granularity, but not grouping strategy, significantly affected execution time.

6.2.8 Discussion

Our results strongly support our hypothesis that granularity significantly affects the rate of fault detection that can be achieved by prioritization techniques. However, we could not distinguish any significant effect caused by test case grouping.

More important from a practitioner's perspective, however, are implications of these results for tradeoffs and factors involved when designing test suites and choosing granularities. We now discuss those implications that we can draw from our

Variable: *Testing time.*					
Source	SS	D.F.	MS	F	p
Program	3130000	1	3130000	157.50	*0.00*
Granularity	7840000	6	1310000	65.70	*0.00*
Grouping	70900	1	70900	3.60	0.06
Program*Granularity	66800	6	11100	0.60	0.76
Program*Grouping	60700	1	60700	3.10	0.08
Granularity*Grouping	34400	6	5730	0.30	0.94
Error	4569911	230	19869		

TABLE 6.5: Execution Time Anova

empirical data, to help clarify the practical impact of the results (taking into consideration the threats to validity discussed in Section 6.2.6).

6.2.8.1 Reducing the Test Suite versus Reducing Overhead

Coarser granularity can greatly increase the efficiency of a test suite. For example, increasing granularity from G1 to G4 on the `emp-server` test suite saved an average of 360 minutes (70% time reduction) in test execution time. The same granularity increase on `bash` saved 390 minutes (50% time reduction) in test execution time.

These differences can be attributed primarily to the amount of overhead in test suite execution required for each program. In our experiments, the savings generated by increases in granularity resulted primarily from reduction in the overhead associated with test setup and cleanup. (In other cases, another factor in overhead might be the cost of human intervention.) Test suites with larger granularity had

fewer test cases, which reduced the overall overhead of the suites; this effect was more profound for `bash`, whose test cases carried more overhead than the `emp-server` test cases. Note, however, the other side of the tradeoff: test suites with low overhead are not likely to yield time savings through increases in granularity.

6.2.8.2 Creating More Prioritizable Test Suites

We have found that in addition to technique and program, granularity has a significant impact on a test suite's rate of fault detection. Finer test suite granularity is likely to result in greater opportunities for prioritization and translate into higher APFD values. When larger test cases are partitioned into smaller ones, the scope of the average test case decreases, allowing prioritization techniques to more precisely discriminate between test cases. Large test cases, even in small numbers, can limit the opportunities for prioritization.

Tradeoffs with execution time, however, may play a part in decisions about granularity, so it is important to note that these results differ with the difficulty of detecting the faults occurring in programs. Programs for which the number of fault exposing test cases is small are more likely to suffer APFD losses from increases in granularity than programs for which the number of fault exposing test cases is large. Most faults in `bash` were relatively difficult to expose: 99% were revealed by fewer than 1% of the G1 test cases. For `emp-server`, in contrast, only 33% of the faults were exposed by fewer than 1% of the G1 test cases; the other 67% of the faults being exposed more often. This result is most evident when considering the APFD results for fn-cov-fb, and for the functional test input grouping. Here, the APFD for `bash` was reduced by 37 points when going from G1 to G64, whereas

the APFD for `emp-server` (which had fewer hard faults) was reduced by only 6 points.

One implication of these results pertains to testing processes, which are typically driven by tradeoffs between the expense of testing and the need to detect faults. When running test cases during development (especially as in test-driven development processes, or test-every-night processes), where initial, easier-to-find faults might be expected to be common, coarse-grained test cases that run faster due to lower setup time requirements may be most cost-effective. When running system tests at the end of development cycles, where the probabilities of individual test cases failing are smaller and the testing interval may be somewhat longer, fine-grained test cases may be most cost-effective.

6.2.9 Conclusion

Writers of testing textbooks have long shown awareness that the composition of test suites can affect the cost-effectiveness of testing. These effects can begin when testing the initial release of a system, where success in finding faults in that release, as well as the amount of testing that can be accomplished, can vary based on test suite granularity and test input grouping. Software that succeeds, however, subsequently evolves: the costs of testing that software are compounded over its lifecycle, and the opportunity to miss faults through inadequate regression testing occurs with each new release. It is thus imperative that researchers study the effects of test suite design across the entire software lifecycle.

6.3 Effects of Changes

6.3.1 Introduction

To create a new software release, engineers apply modifications to the software.[4] These modifications can range in size from tiny bug fixes involving specific statements to complete code redesign involving many modifications. They can also be localized in a few functions or distributed across hundreds of functions. (For example, the *empire* and *bash* program subjects, described earlier in this thesis, have versions with as few as 9 modified functions and versions with as many as 249 modified functions). Finally, modified code can be executed by just a few test cases, or executed by most of the test cases, in a test suite.

In Section 6.2, we studied the effects of test case granularity on prioritization techniques' effectiveness. That study, however, did not specifically examine how the type and magnitude of changes and their relation to test coverage patterns affect the effectiveness of prioritization techniques.

In this section, we present the results of an embedded multiple case study designed to investigate these issues, by observing the application of prioritization techniques to several releases of four software systems.

6.3.2 Experiment Subjects

Several releases of four non-trivial C programs, `bash`, `grep`, `flex`, and `gzip`, were studied. `Bash` is a complete and complex Unix shell, `grep` searches input

---

[4] Portions of this section have appeared previously in [33].

Version	*Functions*	*Changed Functions*	*Lines of Code*	*Regression Faults*
1.0.7	86	–	4,744	-
1.1.2	86	37	5,228	5
1.2.2	103	26	5,811	3
1.2.3	102	14	5,727	2
1.2.4	102	32	5,810	2
1.3	108	52	6,582	3

TABLE 6.6: `Gzip` Experiment Subject

files for a pattern, `flex` is a lexical analyzer generator, and `gzip` is a compression and decompression utility. The `bash` subject was described in Section 5.3.2, and the `flex` and `grep` subjects were described in Section 2.5.1.

The `gzip` subject has not previuosly been used. Table 6.6 lists, for each version of `gzip`, the numbers of functions, changed functions (functions modified or added to the version since the preceding version, or deleted from the preceding version), non-comment, non-blank lines of code (referred to in the rest of this chapter as "executable lines of code"), and seeded regression faults.

`Gzip` subject preparation, test suite creation, fault seeding, and instrumentation were performed by the same processes used in preparing `flex` and `grep`, described in Section 2.5.1.

6.3.3 Empirical Study Design

The overall goal of this study was to determine how the size, distribution, and location of the modifications made to a software system during maintenance affect the effectiveness of prioritization techniques.

To perform the study, several empirical approaches were considered. A case study rather than a formal experiment was selected, because (1) it is not possible to control for the evolution of (or modifications made to) the subject programs studied, or reproduce that evolution at will, and (3) the nature of the questions addressed, which concern how changes impact regression testing methodologies and why technique performance varies under different types of change, are appropriate for a case study [223]. The resulting study employed a multiple-case study design [223], in which each program was studied and analyzed independently (Section 6.3.4). Then, to render the overall results more robust, repeating trends across the four cases (Section 6.3.4.5) were examined.

6.3.3.1 Methodologies and Techniques

Four prioritization techniques were studied, including a control technique, two practical heuristics, and an optimal technique. This choice of heuristics was motivated by the desire to capture a representative sample of the techniques previously presented in this dissertation, currently available to practitioners, and applicable to the subject programs studied. The techniques selected were: random prioritization, function coverage no feedback (fn-cov-nofb), function coverage feedback (fn-cov-fb), and optimal prioritization. These techniques are described in detail in Section 2.3.

6.3.3.2 *Independent Variables*

For each program considered there were two independent variables: the chosen prioritization technique and the program version with its particular changes. The techniques employed for prioritization were just described; the second independent variable, change, was quantified by six metrics that measure change along three dimensions: size, distribution, and coverage, as follows:

- *P-CH-L: percentage of changed lines of code.* P-CH-L measures the fraction of the total executable lines of code that were added, changed, or removed across all functions between a version and its predecessor, and indicates the size of the change made to that version.

- *P-CH-F: percentage of changed functions.* P-CH-F measures the fraction of the total number of functions that contained at least one line changed between a version and its predecessor. To reduce a possible confounding influence caused by test suite coverage, only the changed functions that are present in both versions and that are covered by the test suite were counted. A higher value for P-CH-F indicates higher change distribution, and possibly higher change size.

- *A-CH-LOC-F: average number of lines of code changed per function.* A-CH-LOC-F measures the average amount of change per function in a given version, obtained by dividing the number of changed lines of code by the number of changed functions. High values of this metric indicate larger changes per function.

- *P-CH-Files: percentage of changed files.* P-CH-Files measures the fraction of the total number of files that changed in a given version. A file is considered to have been changed if at least one constituent function changed from the previous version. This value is a measure of change distribution. If a majority of files are changed in a version, it is likely that change propagated across functionalities. The assumption underlying this metric is that functions are grouped together in files based on the functionality they provide.

- *A-Tests-CCHF: average percentage of test cases executing changed functions.* A-Tests-CCHF measures the average fraction of test cases covering the changed functions. Once again, only changed functions that are covered by the test suite were considered. A high value for this metric is likely to reflect changes lying on common test case execution paths. In other words, it implies that functionality that is quite popular with the test suite has been changed. A low value for this metric means that the change is in rarely tested functionality.

- *P-CH-Index: probability of execution of changed functions.* P-CH-Index measures change "popularity"; its value is computed by summing the execution probabilities of changed functions. The execution probability of each function is computed by counting the execution frequency of each function when the test suite is executed. The value of this variable increases as more functions with high execution likelihood are changed.

	Bash		Grep		Flex		Gzip	
	mean	stdev	mean	stdev	mean	stdev	mean	stdev
P-CH-L	2.5	2.2	9.0	9.5	11.6	11.9	16.1	9.7
P-CH-F	12.9	10.6	45.4	33.0	26.2	22.5	33.9	14.0
A-CH-LOC-F	8.1	5.1	19.9	14.6	23.2	11.7	24.0	10.3
P-CH-Files	34.4	19.5	4.8	2.0	3.9	1.6	7.4	4.0
A-Tests-CCHF	69.3	8.6	39.7	3.8	32.6	3.1	6.2	1.1
P-CH-Index	13.8	14.4	49.3	32.6	28.1	22.6	25.2	8.6

TABLE 6.7: Basic Statistics for Change Variables

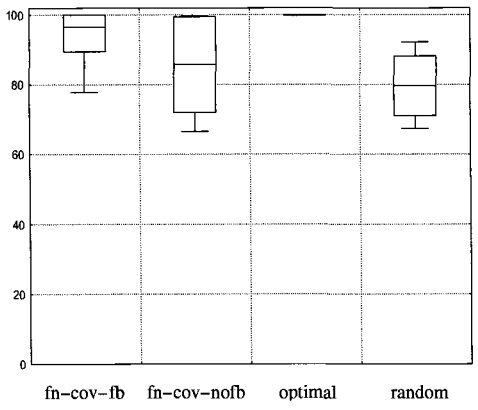
6.3.3.3 Dependent Variable

One dependent variable was measured: savings due to increases in the rate of fault detection, as tracked by APFD (Section 1.2.5).

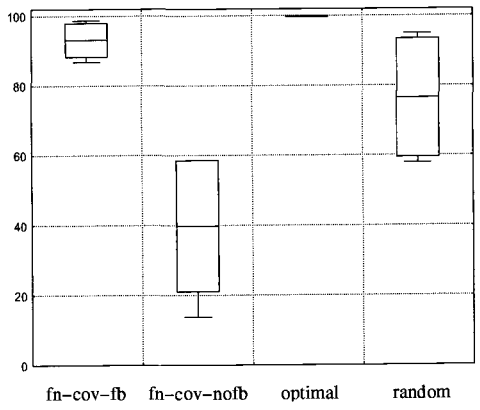
6.3.4 Results and Observations

In this section, the results and observations for each of the units of analysis in the study are presented first individually, and then (Section 6.3.4.5) across those units. Later, Section 6.3.5 explores the practical implications of these observations, and relates them to the questions addressed.

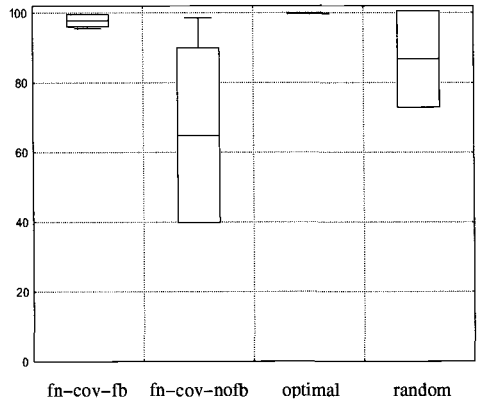
Each of the analyses that follow references the data shown in Table 6.7 and Figure 6.4. Table 6.7 presents the six independent change metrics collected, with their means and standard deviations, for each of the objects of study. Figure 6.4 displays the prioritization results for each of the four observed programs. The results are presented through a series of four box plot diagrams (one per program). The diagrams



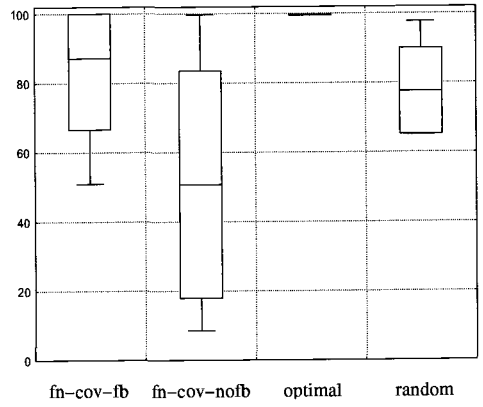
(a) Bash



(b) Grep



(c) Flex



(d) Gzip

FIGURE 6.4: Distribution of APFD variable across versions, per program.

show the distribution of the dependent variable, APFD, plotted separately for each prioritization technique. The mean is used as the measure of central tendency, the

standard deviation is used to represent the variation that encloses each box, and the whiskers are used to represent the range of observed values.

6.3.4.1 Bash

(Figure 6.4(a)). As expected, the optimal technique performed extremely well (APFD over 99% for all versions), with almost no variation in spite of the different size, distribution, and coverage of changes. Fn-cov-fb was second, achieving APFD above 96% on all but version 1. That particular version displayed a unique combination of change characteristics: a high percentage of changed functions (P-CH-F = 22%) covered by a high number of tests (A-Tests-CCHF = 74%). Since fn-cov-fb assigns lower priority to test cases executing functions already covered, having a large number of tests traversing the changed (and possible faulty) functions was not beneficial. Fn-cov-nofb performed consistently worse then fn-cov-fb, with APFD over 90% on only half of the versions. Furthermore, no obvious relationship could be observed between the change patterns and the performance of fn-cov-nofb.

6.3.4.2 Grep

(Figure 6.4(b)). Optimal prioritization performed as expected, that is, a high APFD and low variation. For this program, fn-cov-fb achieved APFD over 86%. Fn-cov-nofb, on the other hand, presented varied results, for an APFD averaging only 40%. Change distribution clearly limited this technique's performance; for example, for version 4, 90% of the functions were changed, fewer than 3% of the lines of code were changed, and APFD was less than 14%.

6.3.4.3 *Flex*

(Figure 6.4(c)). `Flex` displayed behavior similar to `grep`, with optimal and fn-cov-fb always performing above 95%. Fn-cov-nofb again displayed a lot of variation. Version 4 of `flex` had an APFD of 98%, whereas the other 3 versions averaged an APFD of 53%. Interestingly, those three versions contained changes highly distributed across functions, and many lines of code changed within each changed function. For example, in version 2, 57% of the functions were changed and an average of 33 lines of code were changed per function, and here APFD was 42%. This confirms the previous observations that type of change can greatly limit the effectiveness of fn-cov-nofb because prioritizing test cases based on the coverage they provide, without feedback, might ignore the fact that changes (and faults in those changes) are in functions that are rarely covered.

6.3.4.4 *Gzip*

(Figure 6.4(d)). `Gzip` displayed the highest variation for fn-cov-fb and fn-cov-nofb among all subjects. Fn-cov-fb had APFD over 90% on four versions, and 51% on version 3. Version 3 contained the most concentrated changes (P-CH-F 13%) of all versions and the largest number of tests going through those changes (A-Tests-CCHF 8%), which works against the "greedy" nature of fn-cov-fb. Fn-cov-nofb achieved APFD under 60% for four versions. Three of those versions contained large and highly distributed changes, with an average of 30 lines of code changed per function, 41% of functions changed, and over 10% of the files affected by the changes.

6.3.4.5 *Implications Across Case Studies*

The individual patterns described in the previous sections gain significance when observed repeatedly across multiple units of analysis. In this section, a more general data analysis is performed to offer a broader explanation about the sources of similarities and variation across programs, and the impact of change attributes on the cost-effectiveness of prioritization techniques. Two overall implications emerged.

Implication 1: fn-cov-fb always performed well, but fn-cov-nofb was often unpredictable. Fn-cov-fb, on average, always outperformed fn-cov-nofb and random ordering regardless of change characteristics, and also had low variation. For `flex` and `grep`, fn-cov-fb performed close to optimal prioritization on all versions. Fn-cov-nofb displayed large variation, with (for example) APFD ranging from nine to 100 on `gzip`, and performing worse than random prioritization on three of the four programs.

Implication 2: highly distributed changes benefit fn-cov-fb, but can hurt fn-cov-nofb. The difference in performance between the two techniques increased as changes were more distributed. `Grep`, `gzip`, `flex`, and `bash` have decreasing change distribution averages (see Table 6.7), which matches the differences in performance between the two techniques. The two techniques also exhibited similar behavior if a large number of test cases executed the changed functions independent of change distribution. This is evident in all versions of `bash`, where almost 70% of the tests execute the changed functions.

6.3.5 Discussion

This section has presented the results of an embedded multiple case study of the impact of change patterns on the effectiveness of several prioritization techniques.

Overall, the results of this study confirm that change attributes play a significant role in the performance of prioritization techniques; this holds for all the heuristics that were investigated. More important from a practical perspective, however, are the ways in which engineers unaware of this role can make poor choices in designing regression test suites, building modifications into new releases of software, and selecting integration or regression testing strategies and techniques. Even more valuable are the ways in which knowledge of this role can enable better choices.

For example, change size does not seem to be the predominant factor in determining the effectiveness of techniques. Instead, the distribution of changes across functions and files, and whether the test cases reached those changes, seem to be the main contributors to the variation observed across all programs. A simple lines-of-code mentality used to evaluate prospective modifications will not produce effective results.

If a practitioner distributes multiple modifications across many functions and features, employing fn-cov-fb is highly recommended since it seems highly likely to perform close to optimal prioritization, discovering a large percentage of the faults early in the test cycle. On the other hand, fn-cov-nofb performs best if changes occur within the same feature, but even then its results are often discouraging.

6.4 Conclusions

The empirical studies described in this chapter lay the groundwork for developing practical guidelines to improve prioritization effectiveness, by exploring factors that affect prioritization effectiveness. Clearly, additional studies are needed, both to explore other factors, and to extend the external validity of our results. We next turn, however, to an illustration of how the number of influencing factors can be used to help practitioners choose appropriate techniques.

CHAPTER 7

CLASSIFICATION

7.1 Introduction

The studies reported in Chapters 1.4, 3, and 5.4 of this thesis have shown that the rates of fault detection produced by prioritization techniques can vary significantly with several factors related to program attributes, change attributes, and test suite characteristics.[1] In several instances, techniques have not performed as expected. For example, one might expect that techniques that take into account the location of code changes would outperform techniques that simply consider test coverage without taking changes into account, and this expectation is implicit in the technique implemented at Microsoft [185]. Our empirical studies described in Chapter 1.4, however, sometimes revealed results contrary to this expectation. It is possible that engineers choosing to prioritize for both coverage and change attributes may actually achieve poorer rates of fault detection than if they prioritized just for coverage or did not prioritize at all.

More generally, to use prioritization cost-effectively, practitioners must be able to assess which prioritization techniques are likely to be most effective in their particular testing scenarios, i.e., given their particular programs, test cases, and modifications. Toward this end, we might seek an algorithm which, given various met-

[1] Portions of this chapter have appeared previously in [49, 50].

rics about programs, modifications, and test suites, calculates and recommends the technique most likely to succeed. The factors affecting prioritization success are, however, complex, and interact in complex ways, as shown in the previous chapter, and [46]. We do not possess sufficient empirical data to allow creation of such a general prediction algorithm, and the complexities of gathering such data are such that it may be years before it can be available. Moreover, even if we possessed a general prediction algorithm capable of distinguishing between existing prioritization techniques, such an algorithm might not extend to additional techniques that may be created.

In this chapter, therefore, we report on an alternative approach. Using data obtained from the application of several prioritization techniques to several substantial systems, we compare the performance of several prioritization techniques in terms of effectiveness and show how the results of this comparison can be used, together with cost-benefit threshold information, to select a technique that is most likely to be cost-effective. We then show how an analysis strategy based on classification trees can be incorporated into this approach and used to improve the likelihood of selecting the most cost-effective technique.

Our results provide insight into the tradeoffs between techniques, and the conditions underlying those tradeoffs, relative to the programs, test suites, and modified programs that we examine. If these results generalize to other workloads, they could guide the informed selection of techniques by practitioners. More generally, however, the analysis strategy we use demonstrably improves the prioritization technique selection process, and can be used by researchers or practitioners to evaluate techniques in a manner appropriate to their own testing scenarios.

7.2 Empirical Study

To assess whether, and under which conditions, specific code-based prioritization techniques are preferable to other techniques, and to examine approaches for making such assessments, we performed an empirical study. Results of applying various prioritization techniques to a certain number of versions of several non-trivial medium to large size program subjects were studied by examining the rates of fault detection of the resulting test suites using APFD data. (Earlier studies had already collected a portion of the required data, which we simply reuse. The remaining required data were collected specifically for this study.)

7.2.1 Prioritization Techniques

As target prioritization techniques, we chose four heuristics and two control techniques that could easily be (or have already been) implemented by practitioners, and that allow us to examine two of the key dimensions of differences among techniques: the uses of feedback and information on modifications. (For simplicity and to facilitate comparison, we restricted our attention to function-coverage-based techniques). The four techniques were: function coverage no feedback (fn-cov-nofb), function coverage feedback (fn-cov-fb), function binary diff coverage no feedback (fn-bdiff-cov-nofb), function binary diff coverage feedback (fn-bdiff-cov-fb), random prioritization (random), and optimal prioritization (optimal). These techniques are described in detail in Section 2.3.

Program	Version	Lines of C Code	Regression Faults
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 7.1: Experiment Subjects (*Make*, *Sed*, and *Xearth*)

7.2.2 Subject Programs

To reduce the likelihood that our results would be dependent on a specific set of programs, we used eight C programs with different characteristics as subjects: `empire`, `bash`, `grep`, `flex`, `make`, `sed`, `xearth`, and `gzip`. The `grep` and `flex` subjects are described in Section 2.5.1. The `bash` subject was described in Section 5.3.2. The `empire` subject was described in Section 6.2.3. The `gzip` subject was described in Section 6.3.2. Here, we describe only `make`, `sed`, and `xearth` (see Tables 7.1 and 7.2).

7.2.2.1 Make

Make is a utility that reads a given command file that contains a set of targets, each associated with a set of commands. *Make* checks targets against their dependencies.

Program	Test Suite Size
make	1044
sed	1294
xearth	540

TABLE 7.2: Tests per Subject (*Make*, *Sed*, and *Xearth*)

Given a target, this utility processes its dependencies (other targets or files) recursively, and executes commands associated with a given target if necessary. *Make* is usually used to control building of software consisting of multiple files.

7.2.2.2 Sed

Sed is a stream editor that performs specified editing commands to given files.

7.2.2.3 Xearth

Xearth is a program that displays an image of Earth, including currently light and dark zones, on the X Windows background or saves this image into the specified file. It was developed by Kirk Lauritz Johnson and can be freely obtained.

7.2.2.4 Test Suites

For each medium subject program, a *test suite* was created. First, specification-based tests were created using the category-partition method presented in [149]. Second, the test suites were augmented with additional tests to make them branch adequate.

7.2.2.5 Faults

To obtain regression faults, 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, were asked to study programs and inject faults into modified code, doing it as realistically as possible. Ten regression faults have been seeded. Then, they were activated one-by-one and information about which tests revealed which faults has been collected. Faults, not detected by any test, and faults, detected by more then 20% of tests, have been excluded. Undetected faults do not affect prioritization experiments, and faults detected by too many tests are not particularly interesting, because they would have been detected and eliminated early in the development process by unit testing.

7.2.3 Study of Average APFD Values

For each of our subjects, we retrieved or computed relevant APFD values. Specifically, these included APFD values for our five prioritization techniques applied to each pair of sequential versions of each program subject. This yielded 56 APFD values, which are represented in the box-plots shown in Figure 7.1. The figure contains separate plots for each program and one plot summarizing "all programs" (bottom-right). Each plot contains a box showing the distribution of APFD scores for each of the five techniques, and the optimal APFD scores possible.

Overall, the APFD results shown in Figure 7.1 are similar to those observed in earlier studies. Techniques using feedback (fn-cov-fb and fn-bdiff-cov-fb) usually produced better prioritization results than random, and in some cases (e.g., `flex`) approximated optimal ordering. In contrast, the simplest prioritization technique,

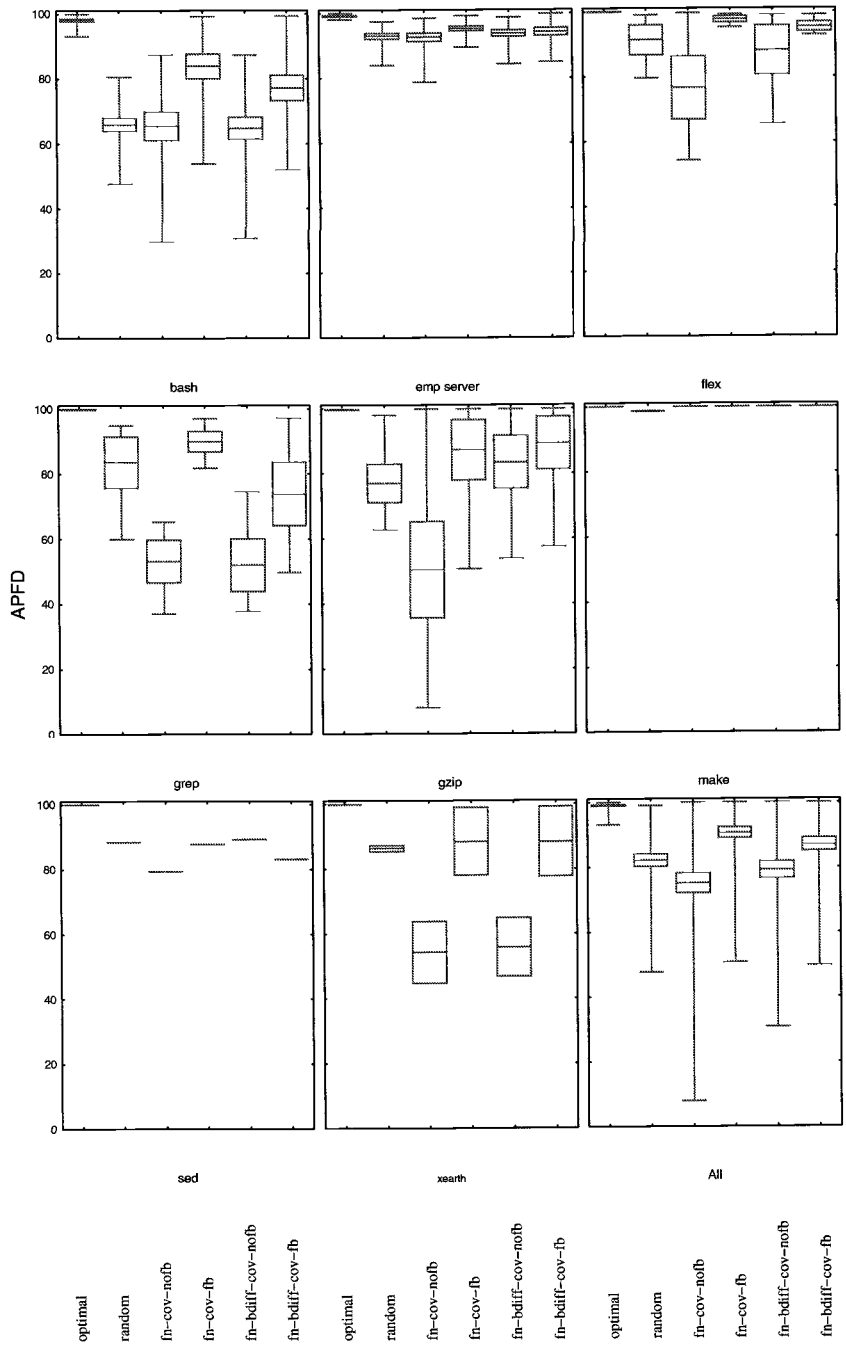


FIGURE 7.1: Average APFDs per technique, and optimal APFD, per program and overall.

fn-cov-nofb, produced an average APFD lower than or equal to that produced by random. Considering the data for all subjects, we note that techniques without feedback tended to have lower APFD values and exhibit greater variance in APFD than techniques with feedback. We also found that the use of modification information (indicated by **bdiff** in the technique's name) sometimes improved the fn-cov-nofb technique (e.g, on `flex`, `gzip`, `sed`, and `xearth`), but often caused the fn-cov-fb technique to behave poorly.

Figure 7.1 also shows the degree to which APFD values varied across subjects and versions. For example, on some subjects (e.g., `gzip`) there was large variance in APFD values for all techniques, while on others (e.g., `make`) APFD values were relatively consistent for all techniques. On still other subjects (e.g., flex), some techniques exhibited wide variance while others did not. The relative performances of techniques also differed across subjects; for example, on `grep`, the mean APFD value for fn-cov-fb was 20 points greater than the mean APFD value for fn-bdiff-cov-fb, while on `gzip`, the mean APFD value for fn-bdiff-cov-fb was better than the mean APFD value for fn-cov-fb.

7.2.4 Study of Prioritization Instances and Cost-Benefit Thresholds

A practitioner turning to Figure 7.1 for help in selecting a prioritization technique could easily be misled. For example, although fn-bdiff-cov-nofb's mean APFD was worse than random's mean APFD on most programs, on 56% of all individual applications fn-bdiff-cov-nofb was superior to random. In general, measures of central tendency such as the mean are appropriate to characterize an aspect of a

distribution, but they do not provide a way to characterize how likely we are, in selecting a technique, to be correct in our selection.

A different strategy for assessing the tradeoffs between prioritization techniques, that does provide a way to characterize the likelihood of selecting a technique correctly, can be obtained by comparing the numbers of applications in which the performances of prioritization techniques differ. Toward this end, we define a *prioritization instance* as a single application of a given prioritization technique to a given version and test suite. To compare two prioritization techniques, as an initial strategy, we calculate the number of instances in which the first technique generates a higher APFD than the second.

When making this comparison, however, we also consider an additional factor. A difference in APFD of k% may or may not be practically important to a practitioner, depending on various cost factors associated with the practitioner's testing process. To assume that a "higher" APFD implies a better technique, independent of cost factors, is an oversimplification that may lead to inaccurate choices among techniques. Cost models for prioritization (Chapter 5) can be used to determine, for a given testing scenario, the amount of difference in APFD that may yield desirable practical benefits by associating APFD differences with measurable attributes such as dollar costs. Without constraining our analysis to specific costs, however, we can analyze cost-benefits more generally by using an abstract notion of the amount of difference in APFD that we refer to as a *cost-benefit threshold*: a percentage difference in APFD that must be exceeded in order for that APFD gain to be beneficial.

Table 7.3 compares the performances of the techniques we investigated in our study in terms of prioritization instances, conditioned on several different cost-benefit thresholds. The table displays, for each pairwise technique comparison (one

Row #	Techniques Compared	Cost-Benefit Threshold 0%	1%	5%	10%	25%
1	fn-cov-nofb vs. random	41	39	21	5	2
2	fn-cov-fb vs. random	79	70	37	32	20
3	fn-cov-fb vs. fn-cov-nofb	61	61	52	39	30
4	fn-bdiff-cov-nofb vs. random	57	41	16	9	2
5	fn-bdiff-cov-nofb vs. fn-cov-nofb	55	50	34	16	5
6	fn-bdiff-cov-nofb vs. fn-cov-fb	23	16	7	4	2
7	fn-bdiff-cov-fb vs. random	68	59	30	25	14
8	fn-bdiff-cov-fb vs. fn-cov-nofb	63	63	46	37	14
9	fn-bdiff-cov-fb vs. fn-cov-fb	41	25	7	2	2
10	fn-bdiff-cov-fb vs. fn-bdiff-cov-nofb	50	41	30	25	14

TABLE 7.3: Percentage of Prioritization Instances in which the First Technique Compared is Better than the Second Technique Compared under a Given Cost-benefit Threshold

per row), the percentage of prioritization instances in which each technique was worth applying, across five cost-benefit threshold values (0%, 1%, 5%, 10%, and 25%). Within each comparison, for each cost-benefit threshold k, we list the percentages of prioritization instances in which the first technique of the two compared (the leftmost technique listed in column 2) produced an APFD value exceeding that of the second technique by $k\%$ or more, and thus, should be the preferred technique under threshold k. Put differently, the numbers contained in the table's cells under a given threshold k indicate the probability that the first technique would achieve an APFD $k\%$ better than the second technique, across the instances in which the techniques were applied.

The data in the table serves as the basis for a prioritization technique selection strategy (henceforth referred to as the *basic instance-and-threshold* strategy). For example, considering row 2, a practitioner can claim some confidence that benefits will be obtained, at low cost-benefit thresholds, by employing fn-cov-fb rather than random, because such benefits were observed in 79% of the instances considered for threshold 0%, and 70% of the instances considered for threshold 1%. On the other hand, considering row 9, a practitioner expecting to obtain benefits by incorporating modification information into fn-cov-fb stands a greater chance of being incorrect than correct, for all cost-benefit thresholds, and a practitioner choosing between fn-cov-nofb and random would select appropriately on more occasions by just choosing random.

The table also shows that, when the practitioner becomes more demanding with respect to cost-benefit threshold, the recommended technique shifts. For example (row 2), although fn-cov-fb is preferable to random when thresholds are low (0% or 1%), as cost-benefit threshold reaches 5% there is a larger probability that the use of feedback is not worthwhile. Also, at cost benefit thresholds of 5% or higher, comparing random to any heuristic (rows 1, 2, 4, and 7), there are always more instances in which random is preferable.

It is also interesting to observe how the rates at which percentages change varies across comparisons. For example, the probability that fn-cov-fb is preferable to fn-cov-nofb decreases more slowly, as threshold increases, than does the probability that fn-cov-fb is preferable to random; this indicates a smoother transition between threshold levels.

Finally, although we cannot claim that the particular results presented in the table generalize to other programs, versions, and test suites, with further experimenta-

tion we hope to improve the generality of the information presented. Meanwhile, a practitioner could collect similar data on their own systems and employ the method just described to determine which techniques to employ on those systems in future regression testing efforts.

7.2.5 Improving Technique Selection using Testing Scenario Characteristics

The basic instance-and-threshold strategy for comparing and selecting prioritization techniques is based exclusively on comparisons of the APFD values achieved by two techniques, relative to a given cost-benefit threshold; it is simple and, our data shows, can be effective in some cases. In other cases, however, this strategy may not be helpful. For example, the table reveals that the chance of achieving higher APFDs by adding feedback to the fn-bdiff-cov-nofb technique (i.e, by using fn-bdiff-cov-fb) is 50/50. That is, the practitioner can obtain the same level of certainty that they are choosing the best technique through a coin-toss.

Whether or not this strategy is effective, it seems possible that a second strategy considering the characteristics of the *testing scenario* (the particular program, modifications, and test suite involved), could increase the probability of choosing a technique correctly. We call this second strategy the *enhanced instance-and-threshold* strategy, and to investigate it we followed a two step process. First, we characterized the collected prioritization instances by computing a set of metrics related to testing scenarios. Second, we used these metrics to refine the guidelines for selecting techniques by building classification trees.

7.2.5.1 *Classification Trees and their Application*

Classification trees have been used frequently in previous software engineering research. For example, classification trees have been used to classify modules as fault prone or not fault prone [98] and to predict components for which development effort is likely to be high [154, 181]. In our context, we use classification trees to predict whether a certain testing scenario facilitates the use of a prioritization technique by measuring program, test suite, and change characteristics that hold for that particular scenario. Classification trees can help with this for several reasons. First, unlike more traditional statistical techniques, classification trees are not constrained by the underlying population distribution. Second, their hierarchical nature makes them easy to interpret, which could facilitate their adoption by practitioners. Third, the trees can be decomposed into a set of rules that make the decision process straightforward.

The generation of a classification tree starts from a set of observations that constitute a training or learning set for which a property that must be forecasted is known. For example, for the question of whether feedback is effective, the learning set must include, for each prioritization instance, whether feedback was beneficial or not. In addition, each prioritization instance has a set of associated values corresponding to a number of independent variables. The tree generation process begins by splitting the learning set (starting node) into two subgroups (child nodes). The method we used for splitting, CART (Classification and Regression Trees), uses an "exhaustive search for univariate splits" method for categorical predictor variables [187], in which all possible splits for each predictor variable at each node are examined to find the split producing the largest improvement in goodness of fit. We

employed the Gini goodness of fit measure, which reaches a value of zero when only one class is present at a node and reaches its maximum value when class sizes at the node are equal. The process is repeated recursively for each node until a stopping rule is reached.

Tree evaluation is commonly performed through misclassification rates. A portion of the cases are designated as belonging to the learning sample and the remaining cases are designated as belonging to the test sample. The predictive model defined by the tree can then be developed using the cases in the learning sample, and its predictive accuracy can be assessed using the cases in the test sample.

7.2.5.2 Using Classification Trees

In our application of the approach, we used 25% of our observations as the test sample, which left 42 observations in the learning set. Also, to check the stability of the trees, we used v-random cross validation within the learning sample. This cross validation involved selecting five random subsamples of equal size from the learning sample and computing the classification tree of the specified size five times, each time omitting one of the subsamples from the computation and using that subsample as a test sample for cross-validation. Thus, each subsample was used four times in the learning sample and just once as the test sample.

In addition, we made the following assumptions while generating trees. First, since the prioritization techniques we examined are relatively simple, we assumed that their cost is equivalent. We also assumed the same misclassification cost, independent of the predicted outcome. Second, we assumed that the prior probability of a technique being successful is proportional to the percentage of observations

Metric	Description	Mean	Median	Std. Dev.
A_FSIZE	mean func. size	54	46	19
AN_CHOC	mean number of changed lines per changed func.	9	7	6
P_FCH_C	percentage of func. changed and covered	12	7	15
P_CH_L	percentage of changed LOC	11	3	17
P_TRCHF	percentage of test cases reaching a changed func.	94	100	23
AP_FET	mean percentage of func. executed by a test case	33	36	9
A_TESTS_CHF	mean number of test cases going through changed func.	40	45	20
P_CH_INDEX	probability of executing changed func.	16	9	18

TABLE 7.4: Metrics Collected over the 56 Applications of Prioritization Techniques to our Subject Programs

in the learning set where that technique generated an APFD value greater than its counterpart (e.g., fn-cov-fb is assumed to be successful in 70% of the scenarios for a 1% cost-benefit). Third, splitting on the predictor variables of the learning set continued until each terminal node in the classification tree had no more than 25% of misplaced scenarios (Fact/frac option in Statistica).

The independent variables we considered were obtained from studies by Elbaum et al. in [46] and Section 6.3, which identified and classified the sources of variation observed in the prioritization techniques's effectiveness. Those sources involve program, test suite and change characteristics. The resulting metric set (Table 7.4) is the result of a refinement process in which several of the originally proposed metrics were discarded based on their marginal contribution to the observed variation in APFD.

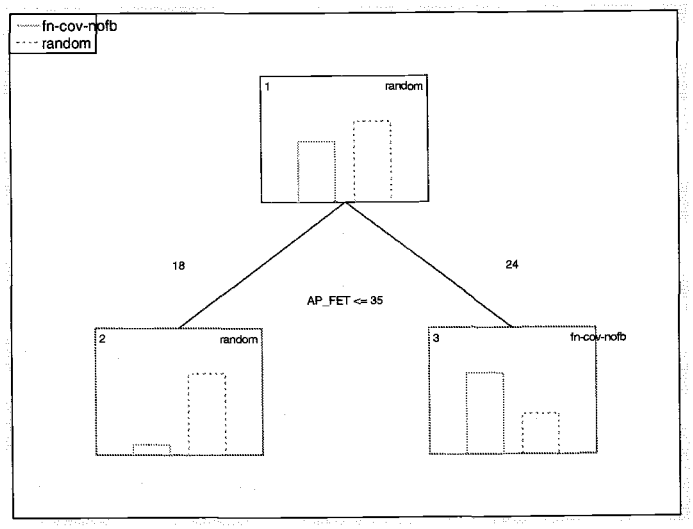


FIGURE 7.2: Classification tree for fn-cov-nofb versus random.

7.2.5.3 Results

We generated classification trees for each pair of techniques compared (each row) in Table 7.3, beginning with the pairs for which the application was successful in producing refinements (rows 1, 2, 3, 5, 8, and 10), followed by the other pairs. (We generated trees only for the cost-benefit threshold of 1%. This choice was arbitrary and does not imply any loss of generality for the approach.)

1. fn-cov-nofb versus random

We begin by comparing fn-cov-nofb and random. Figure 7.2 presents the tree that results from applying the classification tree approach considering these two techniques. The tree starts with the top decision node (node "1"). Each node contains a histogram representing the frequency distribution of the techniques being

		Best	
	Techniques	fn-cov-nofb	random
Predicted	fn-cov-nofb	4	2
	random	0	8
	Misclassification Rates	0/4 = 0%	2/10 = 20%

TABLE 7.5: Classification Accuracy on Test Sample - Fn-cov-nofb versus Random

compared (heights of the columns are proportional to the frequencies); the legend at upper left identifies the techniques to which the bars in the histograms correspond. Each node also contains a label indicating the dominant technique in that node. The root node, node 1, is split forming two new nodes; the text beneath the root node describes the rule determining the split. In Figure 7.2 this rule indicates that instances with AP_FET $\leq$ 35% are sent to node 2 and classified as cases in which random should be used, and instances with AP_FET values greater than 35% are assigned to node 3 and classified as cases in which fn-cov-nofb should be used. The values of 18 and 24 printed above nodes 2 and 3, respectively, indicate the number of cases sent to each of these two child nodes.

The tree indicates that fn-cov-nofb worked best on testing scenarios in which test cases executed relatively larger percentages of the functions in the system tested, leading to a higher probability of covering a changed function. On the other hand, fn-cov-nofb did not perform as well in scenarios in which test cases had smaller "footprints", where the probability of not executing a faulty function is accentuated.

We next employed the test set to assess tree accuracy. The results are presented in Table 7.5. The rows in this table correspond to the technique predictions for the

instances, and the columns indicate actual observed results. For example, in four test instances fn-cov-nofb was the best performer and this was correctly predicted, whereas in two instances our predictions were incorrect because random did better. Overall, we observe that with just one metric, the tree could discern with 100% accuracy the instances in which fn-cov-nofb would be preferable to random, and with 86% accuracy the instances in which fn-cov-nofb would not outperform random.

Recall that a practitioner following the guidelines given in Table 7.3, at cost-benefit threshold 1%, would discard fn-cov-nofb and employ no prioritization technique, missing an opportunity for improvement in 39% of the instances. Following the guidelines presented in the classification tree would increase the likelihood of selecting the appropriate prioritization technique. As seen in Table 7.3, a practitioner employing the tree would select the appropriate technique in 86% of the instances: $(\frac{\text{number of correctly classified instances}}{\text{testing set size}}) * 100\% = 100\% * \frac{12}{14} = 86\%$

2. fn-cov-fb versus random

Figure 7.3 presents the classification tree that results from comparing fn-cov-fb and random. This tree contains two splits. The first split is based on A_FSIZE, indicating that programs with average function size over 90 LOC are less likely to provide instances in which fn-cov-fb is preferable to random. The second split employs the AN_CHOC metric with a value of 20, suggesting that functions in which average change size is greater than 20 reduce the potential of fn-cov-fb in comparison to random. Overall, programs with large functions or functions with many changes tend to constrain the power of fn-cov-fb. This could be caused by the greedy nature of fn-cov-fb and the fact that we employed it at the function coverage level.

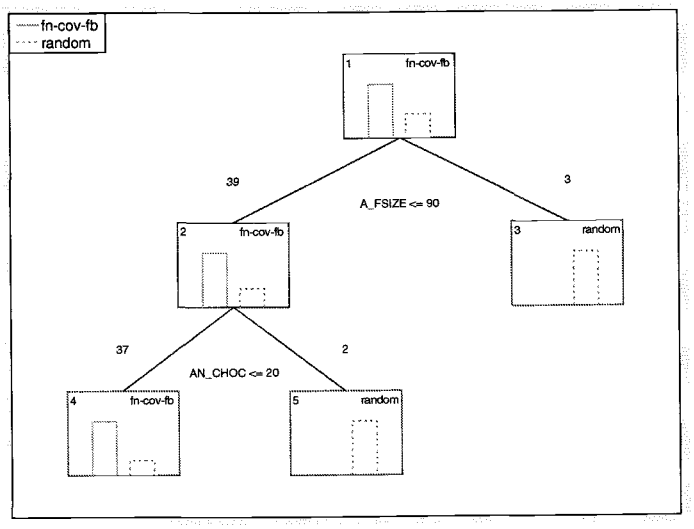


FIGURE 7.3: Classification tree for fn-cov-fb versus random.

		Best	
	Techniques	fn-cov-fb	random
Predicted	fn-cov-fb	10	3
	random	0	1
	Misclassification Rates	0/10 = 0%	3/4 = 75%

TABLE 7.6: Classification Accuracy on Test Sample - Fn-cov-fb versus Random

As such, even if a function has a lot of change or is large, fn-cov-fb seeks to cover it once, which may delay exposure of faults.

Table 7.6 assesses the accuracy of this tree. The misclassification rates indicate that, using the tree, we may over-predict the cases in which fn-cov-fb will be preferable. Still, the guidelines from Table 7.3 (at cost-benefit level 1%) would lead

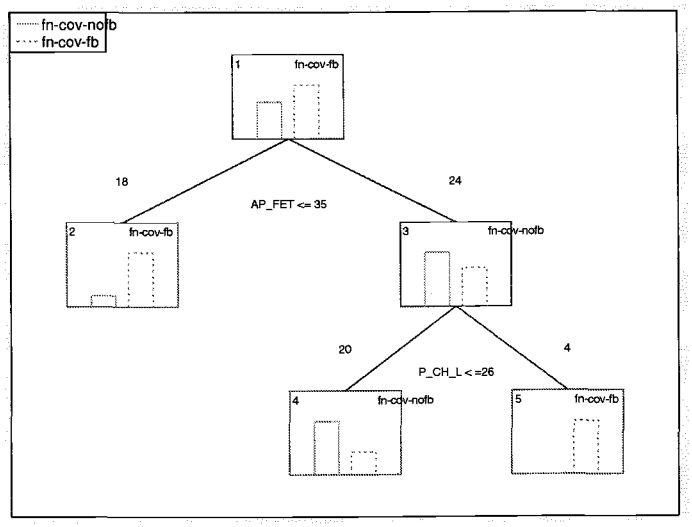


FIGURE 7.4: Classification tree for fn-cov-fb versus fn-cov-nofb.

a practitioner to choose fn-cov-fb in all cases, missing an opportunity to do better in 30% of the instances in which random is at least as good as fn-cov-fb. The refined strategy increases the likelihood of selecting the appropriate prioritization strategy to 79% of the cases ($79\% = 100\% * \frac{11}{14} * 100\%$).

3. fn-cov-fb versus fn-cov-nofb

Figure 7.4 presents the classification tree that results from comparing fn-cov-fb to fn-cov-nofb. This tree also contains two splits. AP_FET is again the first discriminator and, as in the preceding tree, smaller AP_FET values do not benefit fn-cov-nofb. We believe that the availability of test cases focusing on specific functionality (instead of exercising most of the system) are one determinant for whether feedback techniques prosper. Node 3 is split again into two nodes based on the

		Best	
	Techniques	fn-cov-fb	fn-cov-nofb
Predicted	fn-cov-fb	8	1
	fn-cov-nofb	1	3
	Misclassification Rates	1/9 = 11%	1/4 = 25%

TABLE 7.7: Classification Accuracy on Test Sample - Fn-cov-fb versus Fn-cov-nofb

P_CH_L metric. The tree indicates that more changes are likely to help feedback (assuming those changes are distributed).

A practitioner has much to gain by using just two metrics and the classification tree. Without using this information, fn-cov-fb would be selected because it performs better than fn-cov-nofb 61% of the time. Table 7.7 indicates that in this case, a practitioner employing the tree would select the appropriate technique in 86% of the instances ($86\% = 100\% * \frac{12}{14}$).

5. fn-bdiff-cov-nofb versus fn-cov-nofb

Figure 7.5 presents the classification tree that results from comparing fn-bdiff-cov-nofb and fn-cov-nofb. This tree contains three splits. The first split is based on AP_FET, again indicating that higher percentages of functions executed per test case do promote the effectiveness of fn-cov-nofb. Node three is split based on P_CH_L, indicating that fn-cov-nofb is more likely to be beneficial if the percentage of changed lines of code is less than 1%. The last split occurs on node five based on A_TESTS_CCHF. If the percentage of test cases covering changed functions is less than or equal to 46%, then incorporation of diff information seems to be helpful. Intuitively, if the test cases have a greater overlap covering changed functions,

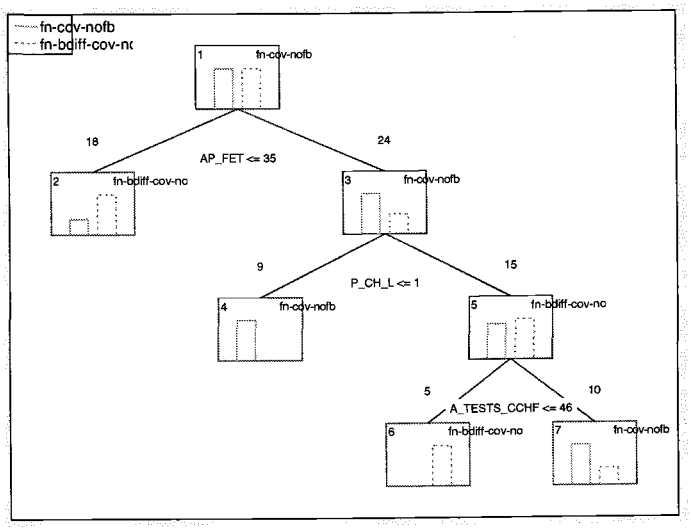


FIGURE 7.5: Classification tree for fn-cov-nofb versus fn-bdiff-cov-nofb.

		Best	
	Techniques	fn-cov-nofb	fn-bdiff-cov-nofb
Predicted	fn-cov-nofb	5	0
	fn-bdiff-cov-nofb	2	7
	Misclassification Rates	2/7 = 29%	0/7 = 0%

TABLE 7.8: Classification Accuracy on Test Sample - Fn-cov-nofb versus Fn-bdiff-cov-nofb

then fn-cov-nofb can potentially do as well as fn-bdiff-cov-nofb since the use of modification information does not add much value.

Table 7.8 indicates that the tree mispredicted 29% of the instances in which fn-bdiff-cov-nofb was not beneficial, but was very accurate in identifying instances

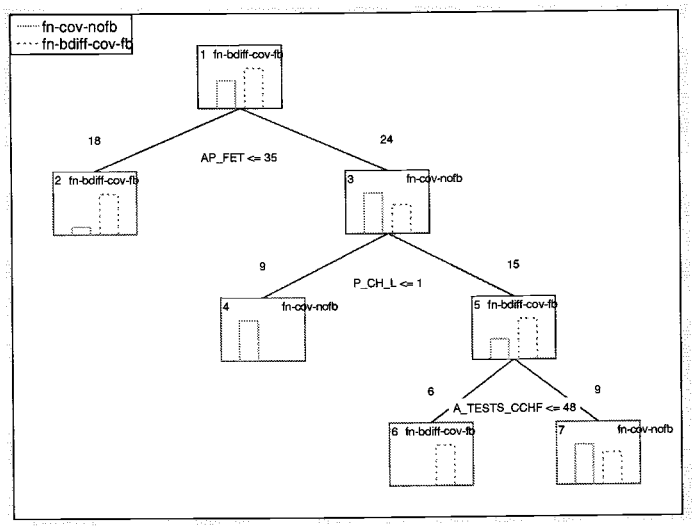


FIGURE 7.6: Classification tree for fn-bdiff-cov-fb versus fn-cov-nofb.

in fn-bdiff-cov-nofb outperforms fn-cov-nofb. A practitioner employing this tree would need to compute three metrics. Such effort would be compensated for, however, with an 86% probability of selecting the appropriate technique ($86\% = 100\% * \frac{12}{14}$). Note that the probability of selecting the appropriate technique without using the tree was 50%.

8. fn-bdiff-cov-fb versus fn-cov-nofb

A practitioner trying to determine whether to incorporate both change information and feedback into fn-cov-nofb would employ the tree in Figure 7.6. The resulting tree has the same nodes and splitting metrics as the one introduced in Figure 7.5.

		Best	
	Techniques	fn-bdiff-cov-fb	fn-cov-nofb
Predicted	fn-bdiff-cov-fb	3	0
	fn-cov-nofb	1	10
	Misclassification Rates	1/4 = 25%	0/10 = 0%

TABLE 7.9: Classification Accuracy on Test Sample - Fn-bdiff-cov-fb versus Fn-cov-nofb

Table 7.9 indicates that a practitioner following this tree would have an 93% probability of selecting the appropriate technique for a given scenario ($93\% = 100\% * \frac{13}{14}$), as opposed to 63% without using the tree.

10. fn-bdiff-cov-fb versus fn-bdiff-cov-nofb

The tree in Figure 7.7 concerns the situation in which a practitioner using modification information must decide whether or not to incorporate feedback. The figure indicates that with just one split based on the P_CH_INDEX metric the leaf nodes are reached. Table 7.10 shows that this tree predicts fn-bdiff-cov-fb to be preferable to fn-bdiff-cov-nofb inaccurately in several instances. Yet, a practitioner following this tree would have a 64% probability of choosing the appropriate technique ($64\% = 100\% * \frac{9}{14}$), which is greater than the 59% chance of choosing correctly without the tree.

Remaining cases.

For four pairs of technique comparisons, classification trees did not provide additional selection power: fn-bdiff-cov-nofb versus random, fn-bdiff-cov-nofb versus fn-cov-fb, fn-bdiff-cov-fb versus random, and fn-bdiff-cov-fb versus fn-cov-fb.

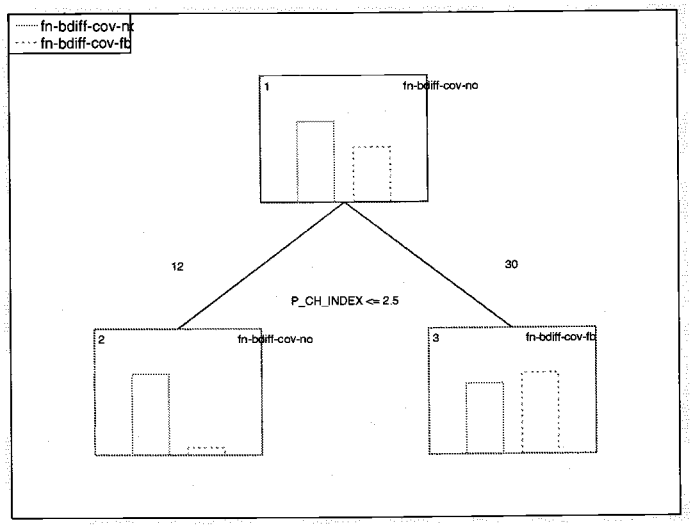


FIGURE 7.7: Classification tree for fn-bdiff-cov-fb versus fn-bdiff-cov-nofb.

		Best	
	Techniques	fn-bdiff-cov-fb	fn-bdiff-cov-nofb
Predicted	fn-bdiff-cov-fb	6	5
	fn-bdiff-cov-nofb	0	3
	Misclassification Rates	0/6 = 0%	5/8 = 63%

TABLE 7.10: Classification Accuracy on Test Sample - Fn-bdiff-cov-fb versus Fn-bdiff-cov-nofb

In these cases, no tree produced gains surpassing those that a practitioner could obtain by following Table 7.3, and thus, trees were of no value. One possible reason for lack of effectiveness in classification trees in these four cases could be that the attributes we captured are not able to explain the differences in technique perfor-

mance. Another possible reason is the limited number of observations available; more observations could improve our understanding and enable the creation of useful classification trees.

It is interesting to note, however, that all comparisons in which a tree could not be constructed involved a comparison with a technique using modification information. Modification information seems, in some cases, to add variability in a manner that we cannot predict. Such variability could be caused, for example, by the accuracy of the tools that determine modifications, or in the way in which modification information is combined with coverage information. Still, it seems that the use of modification information is not always advisable, and its success may be difficult to predict.

7.3 Discussion and Conclusions

We have presented data on the application of five prioritization techniques across eight systems. The performance of these test case prioritization techniques was observed to vary significantly with program attributes, change attributes, test suite characteristics, and their interaction.

These results underscore the need for strategies by which practitioners could choose appropriate prioritization techniques for their particular testing scenarios, and we have proposed two such strategies. The basic instance-and-threshold strategy, introduced in Section 7.2.4, recommends the technique that has been successful in the largest proportion of instances in the past, accounting for cost-benefit thresholds. The enhanced instance-and-threshold strategy, introduced in Section 7.2.5, adds into consideration the attributes of a particular testing scenario, using metrics

Row #	Techniques Compared	Likelihood of Correct Recommendation		Strategy Refinement Gain
		Basic Strategy	Enhanced Strategy	
1	fn-cov-nofb vs. random	61%	86%	25%
2	fn-cov-fb vs. random	70%	79%	9%
3	fn-cov-fb vs. fn-cov-nofb	61%	86%	25%
4	fn-bdiff-cov-nofb vs. random	59%	–	–
5	fn-bdiff-cov-nofb vs. fn-cov-nofb	50%	86%	36%
6	fn-bdiff-cov-nofb vs. fn-cov-fb	84%	–	–
7	fn-bdiff-cov-fb vs. random	59%	–	–
8	fn-bdiff-cov-fb vs. fn-cov-nofb	63%	93%	30%
9	fn-bdiff-cov-fb vs. fn-cov-fb	75%	–	–
10	fn-bdiff-cov-fb vs. fn-bdiff-cov-nofb	59%	64%	5%

TABLE 7.11: Strategies for Prioritization Technique Selection

to characterize scenarios, and employing classification trees to improve the likelihood of recommending the proper technique for each particular case.

The relative effectiveness of these two strategies for a cost-benefit threshold of 1% is summarized in Table 7.11. Each row introduces the techniques being compared, the probability for recommending the appropriate technique for a given scenario under each strategy, and the gain generated by the enhanced strategy with respect to the basic strategy. For example, in the first row, we see that a practitioner employing the basic instance-and-threshold strategy would have a 61% likelihood of selecting the most effective technique. A practitioner using the enhanced strategy, however, would have an 86% likelihood of selecting the most effective strategy (at the cost of computing AP_FET and following the classification tree introduced in

the previous section). The effectiveness of these strategies on the workloads we considered demonstrates their viability for evaluating techniques in other scenarios introduced by researchers or practitioners.

In this work we have assumed that the prioritization techniques examined have equivalent costs. For the relatively simple techniques we have considered, all operating at the level of function coverage and using binary "diff" decisions that could be retrieved from configuration management, this assumption seems reasonable. To extend these comparisons to other classes of techniques, however, this assumption is less reasonable. Techniques that incorporate test cost or module criticality information, or those that operate at finer grained levels of coverage, present different cost-benefits tradeoffs. These tradeoffs can be modeled as described in Chapter 5, and related to cost-benefit thresholds, allowing comparisons of differing-cost techniques, but this approach needs to be investigated empirically.

CHAPTER 8

CONCLUSIONS, CONTRIBUTIONS, AND FUTURE WORK

8.1 Conclusions and Contributions

Before this research began, only three papers [175, 178, 213] had explored test case prioritization. However, recently, additional papers by other researchers and practitioners have been published presenting prioritization techniques. This fact reflects a growing interest in this topic. The research presented in this thesis is the first comprehensive study of test case prioritization, providing valuable information on test case prioritization and its problems. This work can be used as a benchmark for future research in this area, for comparing new techniques and their effectiveness to those presented here. In addition, there are many issues that concern the potential for making prioritization practical. This work resolves several of these issues.

In the following sections, we summarize the contributions and merits of this research.

8.1.1 Prioritization Techniques

We developed a wide variety of prioritization techniques. Researchers can use this set of techniques as a starting point for developing new techniques. Software engineers can choose techniques from these to apply to industrial projects.

8.1.2 *Extensive Studies of Technique Effectiveness*

We have performed extensive studies to assess the effectiveness of prioritization techniques. These studies provide data on techniques to facilitate selection of the most practical ones. These studies also provide a baseline for further empirical work.

8.1.3 *Version-specific and Arbitrary Granularity Level Prioritization Techniques*

We have developed version-specific prioritization techniques that target a given software release, improving, in many cases, prioritization effectiveness. To do this, we incorporated modification information into prioritization techniques. To address the potential cost of low level analyses, we also developed prioritization techniques that utilize arbitrary levels of coverage granularity. Our techniques can be easily scaled up to handle large software systems that may not be practically handled using statement- and branch-level prioritization techniques.

8.1.4 *Cost-cognizance*

With the exception of Wong et al. [213], who briefly mentioned the usage of non-uniform test costs, all prior research on prioritization has assumed uniform test costs and uniform fault severities. We lifted these assumptions by developing the new cost-cognizant $APFD_C$ metric and prioritization techniques. We also suggested several approaches for measuring cost-cognizant data after testing has completed and for estimating the same data before testing starts. This work brings

prioritization one step closer to practical application for cases in which we cannot ignore differences in test costs or fault severities.

8.1.5 Prioritization Framework

We exploited similarities among prioritization techniques to develop a unifying prioritization framework. This framework can express every prioritization technique developed so far. Its main values are its ability to facilitate creation of new prioritization techniques and their analysis and its provision of a standard way of looking at techniques. This framework allows us to implement a general prioritization algorithm whose parameters can instantiate various prioritization techniques. It allows rapid prototyping of and research on a variety of new prioritization techniques with minimal coding, shortening study time, encouraging experimentation with development of new techniques, and reducing the number of errors that might occur if every technique was implemented from scratch.

8.1.6 Cost Model

Previous research assumed that if a prioritization technique achieved a better test case ordering, it would be beneficial. Because there are costs associated with the application of prioritization, this might not always be the case. Thus, we developed a cost model for prioritization. Our model allows us to assess whether to apply prioritization at all, and if so, which technique would be the best. This model is important if prioritization is to be used in practice: the decision to incorporate prioritization into a software maintenance process must be made only if it is economical.

8.1.7 Factors

Studies have shown significant variation among prioritization techniques, applied to different subjects, versions, and test suites. This illustrates that prioritization is sensitive to many factors. Previous studies did not explore such factors, considering techniques' average behavior and ignoring prioritization techniques' performance variations. In our studies, we have isolated some relevant factors and investigated their effects on prioritization. Knowing these factors can be beneficial in several ways. First, they provide an important basis for the theoretical examination of prioritization. If a mathematical model for prioritization is to be constructed, it needs to incorporate factors that affect prioritization. Information on factors and their effects can also be used to develop guidelines for software implementation and maintenance. Adherence to such guidelines may improve prioritization techniques' behavior, making them more effective and predictable, and reducing variation.

8.1.8 Prediction

In previous work, researchers presented techniques and, in some cases, showed which techniques had the best average behavior. However, most prioritization techniques have variations in their performance. As a result, there is no technique that is best all of the time. Being able to predict which technique will be the most beneficial under given circumstances could help practitioners achieve better overall prioritization effectiveness. We have developed an approach for such prediction using classification trees. Studies show that this method can substantially increase the accuracy of prediction of the best technique relative to selecting one based on

its average behavior. Practitioners can use this prediction process to better select prioritization techniques under particular scenarios.

8.2 Future Work

While this research explores test case prioritization and resolves some of its problems and issues to make it more practical and understandable, it also opens up many possibilities for future work.

8.2.1 Experiment Materials

In some of our experiments and case studies, quantitative analyses could not reach statistical significance to reject our null hypotheses due to lack of observations, thus, in these cases, results were inconclusive (for significance level $\alpha = 0.05$). Because there are substantial variations in prioritization effectiveness, the number of observations needs to be large enough for results to stabilize. Larger numbers of subjects and versions can solve this problem.

Several of the studies in this thesis are case studies. The main problem with case studies is that they usually do not produce results that generalize. If studies with representative parameters are used, we can obtain results that could be expected to reoccur under similar circumstances. However, regression testing can have very different characteristics for different scenarios. Practitioners need results that can be generalized over a wide variety of environments. To obtain such results, we need to conduct additional controlled experiments. Such experiments need to control independent variables whose values should be randomly sampled from the pool of

possible values. For "subject" and "version" variables, we need to obtain a large number of subjects and their versions. Another issue is that some independent variables could not be effectively controlled because only one, possibly biased, value for each variable was used. For many subjects, a single test suite per subject was created using a TSL-based testing method and augmented to be branch adequate. While this works for case studies, in order to conduct controlled experiments, we need a random sample of such test suites. This problem could be addressed by creating a large enough test case pool and randomly generating a number of branch adequate test suites to be used in experiments. Seeding a large number of regression faults and randomly sampling several of their subsets would provide an effective control of the "fault set" variable.

Finally, an effective way to study prioritization factors would be to control modifications, test suites, and fault sets by identifying sets of metrics over them, and randomly generating choices according to these metrics.

8.2.2 Cost-cognizance

Our incorporation of cost-cognizance into the APFD metric and prioritization techniques leaves several unresolved issues. The first issue is that there can be many different approaches to fault severity measurement and estimation. We explored only one: using module criticality. This approach is applicable if we want to estimate severities of individual faults related to their consequences. Another approach considers software reliability as the main indicator of fault severity. In this approach, we can consider an operational profile and estimate effects of faults on reliability. For example, if a given fault causes the software to fail 10% of the time, its severity

should be higher than the severity of a different fault which makes the software fail 5% of the time. Thus, this approach uses a fault's effect on software reliability as its severity. This is useful for cases where software failures are not critical and result, for example, in decreased market share. Here, as an estimation of a fault's severity, its connection with test cases can be exploited. Test criticality can be computed and used in prioritization as the effect of the software operations (externally defined functions that the software performs, e.g., "open document", "change font", or "insert table") on software reliability which are exercised by this test case. In other words, a test case has higher criticality if it exercises operations that are more frequently used. Thus, this approach to measurement and estimation of fault severity needs to be studied in detail.

The second issue involves which distributions and scales should be used for cost-cognizance information. Different equivalence classes may be considered for cost-cognizant data. Also, there are many different ways in which to combine cost-cognizant information with other data when prioritizing. In addition to multiplication and tie resolution, we can use other functions including summation, weighted average, geometric mean, exponentiation, and so on. This can potentially generate a large set of prioritization techniques. Alternatively, we can parameterize this function and apply search algorithms to maximize a technique's effectiveness.

A third issue is the need to generalize our results by performing studies on a larger scale.

A fourth issue is that our cost-cognizant $APFD_C$ metric is not the only possibility. We can explore other objective functions on test case orderings in a cost-cognizant environment.

8.2.3 Cost Model

Our cost-benefit tradeoffs model considers one metric evaluating costs saved by applying a prioritization technique: delays. While we believe this is appropriate in many practical cases, it may not always be the best model. The model provides an upper bound on cost savings due to better test case orderings. With further study of prioritization, we could hope to devise alternative metrics that are applicable under other scenarios.

8.2.4 Factors

In Chapter 5.4, we isolated several factors that affect prioritization effectiveness. Because not all variations can be explained by these factors, there must be other factors. Future work could isolate and study the most influential of these factors. This may lay the groundwork for theoretical models of prioritization effectiveness and help us to develop better guidelines for practitioners to use in designing and maintaining software and test suites.

8.2.5 Prediction

If prioritization is to be practically used in the software development environment, prediction of effectiveness for prioritization techniques is essential. Because of substantial variations in performance, accurate prediction can make prioritization more effective. Our study showed that while our predictor achieved significant improvements in accuracy over the method that used the technique that was the best on average, in some cases, it failed to accurately predict the best techniques.

Devising a larger set of metrics on code, changes, and test suites is necessary to make the predictor more accurate.

Classification trees are not the only way to implement a predictor. They rely on an assumption that a single metric's effect can be described as a simple comparison to a threshold value. It is likely that much more complicated relations are taking place. Regression analysis and neural networks could be explored to achieve better prediction accuracy.

8.2.6 Metrics on Test Case Orderings

We used $APFD$ and $APFD_C$ as metrics to assess prioritization effectiveness. There can be many other goals for prioritization such as faster coverage, faster modification coverage, faster reliability estimation, and so on. These metrics can be explored in ways similar to those that we used to explore $APFD$ and $APFD_C$.

8.2.7 Combining Prioritization with Other Regression Testing Techniques

Some industrial software development projects employ regression test selection and test suite reduction. It can be beneficial to keep these techniques and introduce prioritization as part of the regression testing process. Future research could include combining prioritization with regression test selection and test suite reduction to combine the benefits of all these techniques. An approach suggested by Wong et al. in [213] used prioritization as part of regression test suite selection where the first n test cases in the prioritized test suite were used in testing.

BIBLIOGRAPHY

[1] H. Agrawal, J. Horgan, E. Krauser, and S. London. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance*, pages 348–357, September 1993.

[2] IEEE Standards Association. *Software Engineering Standards*, volume 3 of *Std. 1061: Standard for Software Quality Methodology*. Institute of Electrical and Electronics Engineers, 1999 edition, 1992.

[3] IEEE Standards Association. *Software Engineering Standards*, volume 4 of *Std. 1044: Classifications for Software Anomalies*. Institute of Electrical and Electronics Engineers, 1999 edition, 1993.

[4] A. Avritzer and E. J. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Transaction on Software Engineering*, 21(9):705–716, September 1995.

[5] J. Bach. Useful features of a test automation system (part iii). *Testing Techniques Newsletter*, October 1996.

[6] R. Bache. The effect of fault size on testing. *Software Testing, Verification and Reliability*, 7:139–152, 1997.

[7] A. L. Baker, J. M. Bieman, N. Fenton, D. A. Gustafson, A. Melton, and R. Whitty. Philosophy for software measurement. *Journal of Systems and Software*, 12(3):277–281, 1990.

[8] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. In *Proceedings of the 3rd Symposium on Software Testing, Analysis, and Verification*, pages 210–218, December 1989.

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

[10] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, January 1993.

[11] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, NY, 1990.

[12] B. Beizer. *Black-Box Testing*. John Wiley and Sons, New York, NY, 1995.

[13] P. Benedusi, A. Cimitile, and U. De Carlini. Post-maintenance testing based on path change analysis. In *Proceedings of the Conference on Software Maintenance*, pages 352–361, October 1988.

[14] J. Bible, G. Rothermel, and D. S. Rosenblum. A comparative study of coarse- and fine-grained safe regression test selection techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):149–183, April 2001.

[15] R. Binder. *Testing Object-Oriented Systems*. Addison Wesley, Reading, MA, 2000.

[16] R. V. Binder. Testability. *Communications of the ACM*, 37(9):87–101, September 1994.

[17] D. Binkley. Using semantic differencing to reduce the cost of regression testing. In *Proceedings of the Conference on Software Maintenance*, pages 41–50, November 1992.

[18] D. Binkley. Reducing the cost of regression testing by semantics guided test case selection. In *Proceedings of the Conference on Software Maintenance*, October 1995.

[19] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8), August 1997.

[20] D. Binkley. Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. In *Proceedings of the International Conference on Software Engineering*, pages 352–355, April 1998.

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

[22] P. A. Brown and D. Hoffman. The application of module regression testing at TRIUMF. *Nuclear Instruments and Methods in Physics Research*, Section A, .A293(1-2):377–381, August 1990.

[23] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18:31–45, 1982.

[24] John M. Chambers, William S. Cleveland, Beat Kleiner, and Paul A. Tukey. *Graphical Methods for Data Analysis*. Wadsworth International Group, Belmont, CA, 1983.

[25] T. Y. Chen and M. F. Lau. Dividing strategies for the optimization of a test suite. *Information Processing Letters*, 60(3):135–141, March 1996.

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

[27] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and Wong M. Y. Orthogonal defect classification - a concept for in-process measurement. *IEEE Transactions on Software Engineering*, 18(11):943–956, November 1992.

[28] M. Davis, R. Sigal, and E. Weyuker. *Computability, Complexity, and Languages*. Academic Press, Inc., Boston, MA, second edition, 1994.

[29] Márcio E. Delamaro and José C. Maldonado. Proteum–A Tool for the Assessment of Test Adequacy for C Programs. In *Proceedings of the Conference on Performability in Computing Systems (PCS 96)*, pages 79–95, July 1996.

[30] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, April 1978.

[31] T. Dogsa and I. Rozman. CAMOTE - computer aided module testing and design environment. In *Proceedings of the Conference on Software Maintenance*, pages 404–408, October 1988.

[32] S. Elbaum, P. Kallakuri, A. Malishevsky, G. Rothermel, and S. Kanduri. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. Technical Report 020701, University of Nebraska - Lincoln, July 2002.

[33] S. Elbaum, P. Kallakuri, A. G. Malishevsky, G. Rothermel, and S. Kanduri. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Journal of Software Testing, Verification, and Reliability*, 13(2), June 2003.

[34] S. Elbaum, A. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Proceedings of the International Symposium Software Testing and Analysis*, pages 102–112, August 2000.

[35] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. Technical Report 00-60-05, Oregon State University, August 2000.

[36] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. Technical Report 00-60-03, Oregon State University, February 2000.

[37] S. G. Elbaum. *Conceptual framework for a software black box.* Ph.D. dissertation, University of Idaho, July 1999.

[38] S. G. Elbaum and J. C. Munson. A standard for the measurement of C complexity attributes. Technical Report TR-CS-98-02, University of Idaho, February 1998.

[39] S. G. Elbaum and J. C. Munson. Code churn: A measure for estimating the impact of code change. In *Proceedings of the International Conference Software Maintenance*, pages 24–31, November 1998.

[40] S. G. Elbaum and J. C. Munson. Getting a handle on the fault injection process: Validation of measurement tools. In *Proceedings of the International Symposium on Software Metrics*, pages 133–141, November 1998.

[41] S. G. Elbaum and J. C. Munson. Intrusion detection through dynamic software measurement. In *Workshop on Intrusion Detection and Network Monitoring*, pages 41–50, April 1999.

[42] S. G. Elbaum and J. C. Munson. Software evolution and the code fault introduction process. *Empirical Software Engineering Journal*, 4(3):241–262, September 1999.

[43] S. G. Elbaum and J. C. Munson. Evaluating regression test suites based on their fault exposure capability. *Journal of Software Maintenance*, 12(3):171–184, 2000.

[44] S. G. Elbaum and J. C. Munson. Investigating software failures with a software black box. In *Proceedings of the IEEE Aerospace Conference*, March 2000 (to appear).

[45] S. G Elbaum, J. C. Munson, and M. Harrison. CLIC: a tool for the measurement of software system dynamics. Technical Report TR-CS-98-04, University of Idaho, April 1998.

[46] Sebastian Elbaum, David Gable, and Gregg Rothermel. Understanding and measuring the sources of variation in the prioritization of regression test suites. In *Proceedings of the Seventh International Software Metrics Symposium*. Institute of Electrical and Electronics Engineers, Inc., April 2001.

[47] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the International Conference on Software Engineering*, pages 329–338, May 2001.

[48] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions of Software Engineering*, 28(2):159–182, February 2002.

[49] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G. Malishevsky. Selecting a cost-effective test case prioritization technique. Technical Report 03-01-01, University of Nebraska – Lincoln, Department of Computer Science and Engineering, January 2003.

[50] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G. Malishevsky. Selecting a cost-effective test case prioritization technique. (under review).

[51] Brian S. Everitt and Graham Dunn. *Applied Multivariate Data Analysis.* Edward Arnold, 1991.

[52] N. Fenton. Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, March 1994.

[53] N. Fenton and L. Pfleeger. *Software Metrics - A Rigorous and Practical Approach.* PWS-Publishing Company, Boston, MA, second edition, 1997.

[54] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[55] K. F. Fischer. A test case selection method for the validation of software maintenance modifications. In *Proceedings of COMPSAC '77*, pages 421–426, November 1977.

[56] K. F. Fischer, F. Raji, and A. Chruscicki. A methodology for retesting modified software. In *Proceedings of the National Telecommunications Conference B-6-3*, pages 1–6, November 1981.

[57] P. G. Frankl, R. G. Hamlet, B. Littlewood, and L. Strigini. Evaluating testing methods by delivered reliability. *IEEE Transactions on Software Engineering*, 24(8), August 1998.

[58] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, pages 154–164, Victoria, British Columbia, October 1991.

[59] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transaction on Software Engineering*, 19(8):774–787, August 1993.

[60] Rudolf J. Freund and Ramon C. Littell. *SAS for Linear Models: a guide to the ANOVA and GLM procedures.* SAS Institute Inc., Cary, NC, 1981.

[61] D. Gable and S. Elbaum. Extension of fault proneness techniques. Technical Report TRW-SW-2001-2, University of Nebraska - Lincoln, February 2001.

[62] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman, New York, 1979.

[63] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineerijng*. Prentice Hall, Upper Saddle River, NJ 07458, 1st edition, 1991.

[64] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173, June 1975.

[65] T. Goradia. Dynamic impact analysis: A cost-effective technique to enforce error-propagation. In *ACM International Symposium on Software Testing and Analysis*, pages 171–181, June 1993.

[66] T. L. Graves, M. J. Harrold, J-M Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *The 20th International Conference on Software Engineering*, April 1998.

[67] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):184–208, April 2001.

[68] R. Gupta, M. J. Harrold, and M. L. Soffa. An approach to regression testing using slicing. In *Proceedings of the Conference on Software Maintenance*, pages 299–308, November 1992.

[69] G. Hall. *Usage patterns: Extracting system functionality from observed profiles*. Computer science, University of Idaho, April 1997.

[70] D. Hamlet and J. Voas. Faults on its sleeve: Amplifying software reliability testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 89–98, June 1993.

[71] R. G. Hamlet. Probable correctness theory. *Information Processing Letters*, 25:17–25, April 1987.

[72] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, SE-3(4):279–290, July 1977.

[73] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transaction on Software Engineering and Methodology*, 2(3):270–285, July 1993.

[74] M. J. Harrold, J. A. Jones, and Lloyd J. Design and implementation of an interprocedural data-flow tester. Technical report, The Ohio State University, Aug 1997.

[75] M. J. Harrold, L. Larsen, J. Lloyd, D. Nedved, M. Page, G. Rothermel, M. Singh, and M. Smith. Aristotle: a system for the development of program-analysis-based tools. In *Proceedings of the 33rd Annual Southeast Conference*, pages 110–119, March 1995.

[76] M. J. Harrold, D. Rosemblum, G. Rothemel, and E. Weyuker. Empirical studies of a prediction model for regression test selection. *IEEE Transactions on Software Engineering*, 27(3):248–263, March 2001.

[77] M. J. Harrold and G. Rothermel. Separate computation of alias information for reuse. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA) 96*, January 1996.

[78] 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, Mar 1997.

[79] M. J. Harrold and M. L. Soffa. An incremental approach to unit testing during maintenance. In *Proceedings of the Conference on Software Maintenance*, pages 362–367, October 1988.

[80] M. J. Harrold and M. L. Soffa. An incremental data flow testing tool. In *Proceedings of the Sixth International Conference on Testing Computer Software*, May 1989.

[81] M. J. Harrold and M. L. Soffa. Interprocedural data flow testing. In *Proceedings of the 3rd Testing, Analysis, and Verification Symposium*, pages 158–167, December 1989.

[82] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.

[83] J. Hartmann and D. J. Robson. Revalidation during the software maintenance phase. In *Proceedings of the Conference on Software Maintenance*, pages 70–79, October 1989.

[84] J. Hartmann and D. J. Robson. RETEST - development of a selective revalidation prototype environment for use in software maintenance. In *Proceedings of the 23rd Hawaii International Conference on System Sciences*, pages 92–101, January 1990.

[85] J. Hartmann and D. J. Robson. Techniques for selective revalidation. *IEEE Software*, 16(1):31–38, January 1990.

[86] R. Hildebrandt and A. Zeller. Minimizing failure-inducing input. In *Proceedings of the International Symposium Software Testing and Analysis*, pages 135–145, August 2000.

[87] D. Hoffman. A CASE study in module testing. In *Proceedings of the Conference on Software Maintenance*, pages 100–105, October 1989.

[88] D. Hoffman and C. Brealey. Module test case generation. In *Proceedings of the 3rd Workshop on Software Testing, Analysis, and Verification*, pages 97–102, December 1989.

[89] J. R. Horgan and S. A. London. ATAC: A data flow coverage testing tool for C. In *Proceedings of the Symposium on Assessment of Quality Software Development Tools*, pages 2–10, May 1992.

[90] R. Horgan, A. P. Mathur, A. Pasquini, and V. J. Rego. Perils of software reliability modeling. Technical Report SERC-TR-160-P, Software Engineering Research Council, Purdue University, February 1995.

[91] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, SE-2(3):208–215, September 1978.

[92] W. E. Howden. The theory and practice of functional testing. *IEEE Software*, 2(5):6–17, September 1985.

[93] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the International Conference on Software Engineering*, pages 191–200, May 1994.

[94] R. Johnson. *Elementary Statistics*. Duxbury Press, Belmont, CA, sixth edition, 1992.

[95] R. A. Johnson and D. W. Wichorn. *Applied Multivariate Analysis*. Prentice Hall, Englewood Cliffs, N.J., 3rd edition, 1992.

[96] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. In *Proceedings of the International Conference on Software Maintenance*, November 2001.

[97] C. Kaner, J. Falk, and H. Q. Nguyeen. *Testing Computer Software*. Wiley and Sons, New York, 1999.

[98] T. Khoshgoftaar, E. Allen, and J. Deng. Using regression trees to classify fault-prone software modules. *IEEE Transactions on Reliability*, 51(4):455–462, 2002.

[99] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, and N. Goel. Early quality prediction: A case study in telecommunications. *IEEE Software*, 13(1):65–71, January 1996.

[100] T. M. Khoshgoftaar and J. C. Munson. A measure of software system complexity and its relationship to faults. In *Proceedings of the International Simulation Technology Conference*, pages 267–272, 1990.

[101] T. M. Khoshgoftaar and J. C. Munson. Predicting software development errors using complexity metrics. *Journal on Selected Areas in Communications*, 8(2):253–261, February 1990.

[102] T. M. Khoshgoftaar, R. M. Szabo, and J. M. Voas. Identifying program modules with low testability using static software product measurements. Technical Report TR-CSE-94-34, Florida Atlantic University, 1994.

[103] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the International Conference on Software Engineering*, May 2002.

[104] J-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test application frequency. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 126–135, June 2000.

[105] R. E. Kirk. *Experimental Design: Procedures for the Behavioral Sciences*. Brooks/Cole, Pacific Grove, CA, 3rd edition, 1995.

[106] E. Kit. *Software Testing in the Real World*. Addison-Wesley, Reading, MA, 1995.

[107] B. Kitchenham, L. Pickard, and S. Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, 11(7):52–62, July 1995.

[108] B. Korel and A. Al-Yami. Automated regression test generation. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 143–151, 1998.

[109] S. Krishnamurthy and A. P. Mathur. On the estimation of reliability of a software system using reliabilities of its components. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 146–155, Albuquerque; NM, November 1997. IEEE Computer Society.

[110] 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.

[111] J. Laski and W. Szermer. Identification of program modifications and its applications in software maintenance. In *Proceedings of the Conference on Software Maintenance*, pages 282–290, November 1992.

[112] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, SE-9(3):347–54, May 1983.

[113] J. A. N. Lee and X. He. A Methodology for Test Selection. *The Journal of Systems and Software*, 13(1):177–185, September 1990.

[114] H. K. N. Leung and L. White. Insights into regression testing. In *Proceedings of the Conference on Software Maintenance*, pages 60–69, October 1989.

[115] H. K. N. Leung and L. White. Insights into testing and regression testing global variables. *Journal of Software Maintenance*, 2:209–222, December 1990.

[116] H. K. N. Leung and L. J. White. A study of integration testing and software regression at the integration level. In *Proceedings of the Conference on Software Maintenance*, pages 290–300, November 1990.

[117] H. K. N. Leung and L. J. White. A cost model to compare regression test strategies. In *Proceedings of the Conference on Software Maintenance*, pages 201–208, October 1991.

[118] R. Lewis, D. W. Beck, and J. Hartmann. Assay - a tool to support regression testing. In *ESEC '89. 2nd European Software Engineering Conference Proceedings*, pages 487–496, September 1989.

[119] D. Libes. *Exploring Expect: A Tcl-Based Toolkit for Automating Interactive Programs*. O'Reilly & Associates, Inc., Sebastopol, CA, November 1996.

[120] M. Lyu. *Handbook of Software Reliability Engineering*. IEEE Computer Society Press, 1995 edition, 1995.

[121] A. Malishevsky, G. Rothermel, and S. Elbaum. Modeling the cost-benefits tradeoffs for regression testing techniques. In *Proceedings of the International Conference on Software Maintenance*, October 2002.

[122] A. von Mayrhauser, R. T. Mraz, and J. Walls. Domain based regression testing. In *Proceedings of the Conference on Software Maintenance - 1994*, pages 26–35, September 1994.

[123] R. A. D. Millo, R. J. Lipton, and A. J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979.

[124] D. C. Montgomery. *Design and Analysis of Experiments*. John Wiley and Sons, New York, fourth edition, 1997.

[125] L. J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990.

[126] J. C. Munson. Software measurement: Problems and practice. *Annals of Software Engineering*, 1(1):255–285, 1995.

[127] J. C. Munson. A software black box recorder. In *Proceedings of the IEEE Aerospace Conference*, pages 309–320, February 1996.

[128] J. C. Munson. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, January 1996.

[129] J. C. Munson. A functional approach to software reliability modeling. In *Proceedings of the Conference on Mathematical and Scientific Computing: Quality of Numerical Software*, pages 61–76, November 1996.

[130] J. C. Munson and S. G Elbaum. Software reliability as a measurement problem. Technical Report TR-CS-97-03, University of Idaho, March 1997.

[131] J. C. Munson and S. G. Elbaum. Software reliability as a function of user execution patterns. In *Proceedings of the Hawaii International Conference on System Sciencethe Hawaii International Conference on System Sciencess*, page 181, January 1999.

[132] J. C. Munson, S. G. Elbaum, R. M. Karcich, and J. P. Wilcox. Software risk assessment through software measurement and modeling. In *Proceedings of the IEEE Aerospace Conference*, pages 137–147, March 1998.

[133] J. C. Munson and T. Khoshgoftaar. Regression modeling of software quality: An empirical investigation. *Journal of Information and Software Technology*, 32(2):105–114, March 1990.

[134] J. C. Munson and T. M. Khoshgoftaar. The relative software complexity metric: A validation study. In *Proceedings of the Software Engineering Conference*, pages 89–102, 1990.

[135] J. C. Munson and T. M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Transaction on Software Engineering*, pages 423–433, May 1992.

[136] J. Musa. Software faults, software failures, and software reliability modeling. *IEEE Software*, 6(2):85–91, February 1996.

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

[138] J. D. Musa. The operational profile in software reliability engineering: an overview. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 140–154, April 1992.

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

[140] J. D. Musa. Software reliability engineering testing. *Computer*, 29(11):61–68, 1996.

[141] J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability, Measurement, Prediction, Application*. McGraw-Hill, New York, 1987.

[142] P. G. Neumann. *Computer Related Risks*. Addison-Wesley Publishing Company, New York, NY, 1995.

[143] A. P. Nikora. *Software system defect content prediction from development process and product characteristics*. Department of computer science, University of Southern California, Los Angeles, CA, 1998.

[144] A. P. Nikora and J. C. Munson. Determining fault insertion rates for evolving software systems. In *Proceedings of the International Symposium Software Reliability Engineering*, November 1998.

[145] A. P. Nikora and J. C. Munson. Software evolution and the fault process. In *Proceedings of the Twenty Third Annual Software Engineering Workshop, NASA/Goddard Space Flight Center*, 1998.

[146] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Ebgineering Methodology*, 5(2):99–118, April 1996.

[147] J. Offutt, J. Pan, and J. M. Voas. Procedures for reducing the size of coverage-based test sets. In *Proceedings of the Twelfth International Conference Testing Computer Software*, pages 111–123, June 1995.

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

[149] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications ACM*, 31(6):676–686, June 1988.

[150] T. J. Ostrand and E. J. Weyuker. Using dataflow analysis for regression testing. In *Sixth Annual Pacific Northwest Software Quality Conference*, pages 233–247, September 1988.

[151] Lyman Ott. *An Introduction to Statistical Methods and Data Analysis*. PWS-Kent Publishing Company, Boston, MA, third edition, 1988.

[152] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[153] P. Piwowarski, M. Ohba, and J. Caruso. Coverage measurement experience during function test. In *Proceedings of the 15th International Conference on Software Engineering*, May 1993.

[154] A. Porter and R. Selby. Evaluating techniques for generating metric-based classification trees. *The Journal of Systems and Software*, 12(3):209–218, 1990.

[155] R. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New York, NY, 1987.

[156] C. Ramey and B. Fox. *Bash Reference Manual*. O'Reilly & Associates, Inc., Sebastopol, CA, 2.2 edition, 1998.

[157] D. Rosenblum and G. Rothermel. A comparative study of regression test selection techniques. In *Proceedings of the 2nd International Workshop on Empirical Studies of Software Maintenance*, October 1997.

[158] D. Rosenblum and G. Rothermel. An empirical comparison of regression test selection techniques. In *Proceedings of the International Workshop for Empirical Studies of Software Maintenance*, pages 89–94, October 1997.

[159] D. Rosenblum and E. J. Weyuker. Predicting the cost-effectiveness of regression testing strategies. In *ACM SIGSOFT '96 Fourth Symposium on the Foundations of Software Engineering*, October 1996.

[160] D. Rosenblum and E. J. Weyuker. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Transaction on Software Engineering*, 23(3):146–156, March 1997.

[161] D.S Rosenblum and E. J. Weyuker. Lessons Learned from Regression Testing Case Study. *Empirical Software Engineering Journal*, 2(2):188–191, 1997.

[162] G. Rothermel. Efficient, effective regression testing using safe test selection techniques. Technical Report 96-101, Clemson University, January 1996.

[163] G. Rothermel. *Efficient, Effective Regression Testing Using Safe Test Selection Techniques*. Ph.D. dissertation, Clemson University, May 1996.

[164] G. Rothermel and M. J. Harrold. A safe, efficient algorithm for regression test selection. In *Proceedings of the Conference on Software Maintenance*, pages 358–367, September 1993.

[165] G. Rothermel and M. J. Harrold. A framework for evaluating regression test selection techniques. In *Proceedings of the 16th International Conference on Software Engineering*, pages 201–210, May 1994.

[166] G. Rothermel and M. J. Harrold. Selecting regression tests for object-oriented software. In *Proceedings of the Conference on Software Maintenance*, pages 14–25, September 1994.

[167] G. Rothermel and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA 94)*, August 1994.

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

[169] G. Rothermel and M. J. Harrold. Experience with regression test selection. In *Workshop on Empirical Studies of Software Maintenance*, November 1996.

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

[171] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions Software Engineering*, 24(6):401–419, June 1998.

[172] G. Rothermel, M. J. Harrold, and J. Dedhia. Regression test selection for C++ programs. *Journal of Software Testing, Verification, and Reliability*, 10(2), June 2000.

[173] 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 Software Maintenance*, pages 34–43, November 1998.

[174] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization. Technical Report 99-60-12, Oregon State University, December 1999.

[175] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: an empirical study. In *Proceedings of the International Conference Software Maintenance*, pages 179–188, August 1999.

[176] Gregg Rothermel, Sebastian Elbaum, Alexey Malishevsky, Praveen Kallakuri, and Brian Davia. The impact of test suite granularity on the cost-effectiveness of regression testing. In *Proceedings of the 24th International Conference on Software Engineering*, pages 230–240, May 2002.

[177] Gregg Rothermel, Sebastian Elbaum, Alexey Malishevsky, Praveen Kallakuri, and Xuemei Qiu. On test suite composition and the design of cost-effective regression test suites. *ACM Transactions on Software Engineering and Methodology*, (under review).

[178] Gregg Rothermel, Ronald H. Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.

[179] S. Schach. *Software Engineering*. Aksen Associates, Boston, MA, 1992.

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

[181] R. Selby and A. Porter. Learning from examples: Generation and evaluation of decision trees for software resource analysis. *IEEE Transactions on Software Engineering*, 14(12):1743–1757, 1988.

[182] B. Sherlund and B. Korel. Modification oriented software testing. In *Conference Proceedings: Quality Week 1991*, pages 1–17, 1991.

[183] B. Sherlund and B. Korel. Logical modification oriented software testing. In *Proceedings of the Twelfth International Conference on Testing Computer Software*, June 1995.

[184] Sidney Siegel and Jr. N. John Castellan. *Nonparametric Statistics For the Behavioral Sciences*. McGraw-Hill Inc., New York, NY, second edition, 1988.

[185] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 97–106, July 2002.

[186] R. Stallman. *Using and porting GNU CC*. Free Software Foundation, Inc., Cambridge, MA, 1990.

[187] Statsoft. Statistica. http://www.statsoftwarecom/exploratory.html.

[188] A. B. Taha, S. M. Thebaut, and S. S. Liu. An approach to software fault localization and revalidation based on incremental data flow analysis. In *Proceedings of the 13th Annual International Computer Software and Applications Conference*, pages 527–534, September 1989.

[189] K. Tewary and M. J. Harrold. Fault modeling using the program dependence graph. In *IEEE International Symposium on Software Reliability '94*, pages 126–135, November 1994.

[190] M. C. Thompson, D. J. Richardson, and L. A. Clarke. An information flow model of fault detection. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 182–192, June 1993.

[191] J. P. Tsai and S. H. Yang. *Monitoring and Debugging of Distributed Real-Time Systems*. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[192] J. Verner and E. Todd. Experiences with the organization and assessment of group software development projects. In B. Z. Barta, S. L. Hung, and K. R. Cox, editors, *Software Engineering Education (A-40), IFIP Transactions*, pages 309–315. Elsevier Science B. V., North Holland, 1993.

[193] J. Voas. PIE: A dynamic failure-based technique. *IEEE Transaction on Software Engineering*, 18(8):717–727, August 1992.

[194] J. M. Voas, L. J. Morell, and K. Miller. Predicting where faults can hide from testing. *IEEE Software*, 8(2):41–48, March 1991.

[195] F. I. Vokolos and P. G. Frankl. Pythia: a regression test selection tool based on textual differencing. In *Proceedings of the 3rd International Conference on Reliability, Quality & Safety of Software-Intensive Systems (ENCRESS '97)*, May 1997.

[196] F. I. Vokolos and P. G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Proceedings of the International Conference Software Maintenance*, pages 44–53, November 1998.

[197] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[198] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[199] D. Werries and J. C. Munson. Measuring software evolution. In *Proceedings of the International Symposium on Software Metrics*, pages 41–51, March 1996.

[200] E. J. Weyuker. The applicability of program schema results to programs. *International Journal of Computer and Information Science*, 8:387–403, 1979.

[201] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 15(4):465–470, 1982.

[202] E. J. Weyuker. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, SE-12:1128–1138, December 1986.

[203] E. J. Weyuker. Empirical techniques for assessing testing strategies. (Panel discussion at the International Symposium on Software Testing and Analysis), August 1994.

[204] E. J. Weyuker. Using failure cost information for testing and reliability assessment. *ACM Transactions on Software Engineering and Methodology*, 5(2):87–98, April 1996.

[205] E. J. Weyuker and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, SE-6(3):236–246, May 1980.

[206] Elaine. J Weyuker. The evaluation of program-based software test data adequacy criteria. *Communications of the ACM*, 31(6):668–675, June 1988.

[207] L. J. White. Software testing and verification. *Advances in Computers*, 26:335–391, 1987.

[208] L. J. White and H. K. N. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *Proceedings of the Conference on Software Maintenance*, pages 262–270, November 1992.

[209] L. J. White, V. Narayanswamy, T. Friedman, M. Kirschenbaum, P. Piwowarski, and M. Oha. Test Manager: a regression testing tool. In *Proceedings of the Conference on Software Maintenance*, pages 338–347, September 1993.

[210] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, April 1971.

[211] S. Wolfram. *Mathematica: A System for Doing Mathematics on a Computer.* Addison-Wesley, Reading, MA, second edition, 1991.

[212] E. W. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test minimization on fault detection effectiveness. In *Proceedings of the International Conference Software Engineering*, pages 41–50, Seattle, Washington, U.S.A., 1995.

[213] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, pages 230–238, November 1997.

[214] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set size and block coverage on the fault detection effectiveness. In *Proceedings of the Fifth International Symposium on Software Reliability Engineering*, pages 230–238, November 1994.

[215] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *Proceedings of the 17th International Conference on Software Engineering*, pages 41–50, April 1995.

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

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

[218] W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini. Test Set Size Minimization and Fault Detection Effectiveness: A Case Study in a Space Application. Technical Report SERC-TR-173-P, Software Engineering Research Council, April 1997.

[219] W. Wu. Safe and precise test case selection in regression testing. (Unpublished paper), January 1995.

[220] W. Yang. Identifying syntactic differences between two programs. *Software—Practice and Experience*, 21(7):739–755, July 1991.

[221] W. Yang, S. Horwitz, and T. Reps. A program integration algorithm that accomodates semantics-preserving transformations. *ACM Transactions on Software Engineering and Methodology*, 1(3):311–54, July 1992.

[222] S. S. Yau and Z. Kishimoto. A method for revalidating modified programs in the maintenance phase. In *COMPSAC '87: The Eleventh Annual International Computer Software and Applications Conference*, pages 272–277, October 1987.

[223] R. K. Yin. *Case Study Research : Design and Methods (Applied Social Research Methods, Vol. 5).* Sage Publications, London, UK, 1994.

[224] M. Zelkowitz and D. Wallace. Experimental models for validating technology. *IEEE Computer*, 31(5):23–31, May 1998.

[225] J. Ziegler, J. M. Grasso, and L. G. Burgermeister. An Ada based real-time closed-loop integration and regression test tool. In *Proceedings of the Conference on Software Maintenance*, pages 81–90, October 1989.

[226] H. Zuse. *Software Complexity: Measures and Methods.* Walter de Gruyter, New York, NY, 1990.

[227] H. Zuse. *A Framework of Software Measurement*. Walter de Gruyter, New York, NY, 1998.

[228] S. H. Zweben, W. D. Heym, and J. Kimmich. Systematic testing of data abstractions based on software specifications. *Journal of Software Testing, Verification and Reliability*, 1(4):39–55, 1992.

APPENDICES

CHAPTER A

GLOSSARY OF PRIORITIZATION TECHNIQUES

This thesis describes a large number of prioritization techniques. To help identify them, we describe our naming convention for them. Each technique name consists of the following parts: <level>[–<modification information>]–<coverage information>[–<cost cognizance information>]–<feedback>. For each value of each field name we provide a mnemonic, as shown in Table A.1.

Level specifies the granularity level on which data are collected and a given technique operates. For example, information can be collected for each statement and a technique can use this information; such a technique is said to operate at statement level.

Modification information specifies the modification data that are used by a given prioritization technique. These data can be fault index, binary fault index, diff, binary diff, or none (empty field). Types of modification information are described in Section 2.2. When discussing techniques without regard to modification type, we use the term *modification* (mnemonic *mod*) for this field.

Coverage information specifies the way in which coverage data are used in a given prioritization technique. These data can be coverage, fault-exposing-potential, fault-exposing-potential multiply, or fault-exposing-potential double sort. Fault-exposing-potential is described in Section 1.2.6.5. Fault-exposing-potential multiply and fault-exposing-potential double sort are specified when modifica-

level	modification information	coverage information	cost cognizance information	feedback
statement (st)	fault index (fi)	coverage (cov)	none	feedback (fb)
function (fn)	binary fault index (bfi)	fault-exposing-potential (fep)	multiply (ccmult)	no feedback (nofb)
	diff (diff)	fault-exposing-potential multiply (fepm)	award first (cca1st)	
	binary diff (bdiff)		ratio first (ccr1st)	
	none ()			

TABLE A.1: Composition of Prioritization Technique Names

tion information is used. Fault-exposing-potential multiply describes techniques that combine modification and fault-exposing-potential information by multiplying them. Fault-exposing-potential double sort describes techniques that combine modification and fault-exposing-potential information by using modification information to sort award values and fault-exposing-potential information to resolve ties. We shortened the "fault-exposing-potential double sort" name to "fault-exposing-potential". Coverage data are binary values specifying whether a given test case executes a given location.

Cost cognizance information is data relevant to the use of varying test costs and fault severities, as described in Chapter 3. These can involve module criticalities, test costs, and test criticalities. In general, all cost cognizant information can be

present, and, as defined in Section 3.5, all techniques are able to use it. If a technique is not cost-cognizant, it can be described as a special case of a cost-cognizant technique in which we supply unit values instead of costs and criticalities. The **cost cognizant information field** can be <empty> (non-cost cognizant), multiply, award first, or ratio first. These names describe the criticality/cost/award value combination function, presented in Section 3.5.

Feedback specifies whether a technique uses an iterative approach. With **no feedback**, all award values (measuring individual test case `quality`), are computed once and test cases are sorted using these award values. With **feedback**, every time a test case is chosen to be appended to a prioritized test suite, award values are recomputed and data structures are updated.

Not all combinations of fields are meaningful. The **coverage information** field cannot be "fault-exposing-potential multiply" when the **modification information field** is "none".

CHAPTER B

DETAILED DATA FOR STUDIES OF TECHNIQUES INCORPORATING ARBITRARY GRANULARITY AND CHANGE INFORMATION (CHAPTER 1.4)

The following tables present results of Anova analysis and Bonferroni grouping for prioritization technique performance in terms of APFD values. These tables contain information presented and discussed in Section 2.4 in Chapter 1.4. Essentially, these tables augment tables from Section 2.4 presenting data on a per-program basis. (Tables in Section 2.4 present the data averaged for all programs.)

	SS	Degrees of freedom	MS	F	p
TECHN	21867583	4	5466896.00	11014.200	0.00
Error	1661779	3348	496.35		

(a) print_tokens

	SS	Degrees of freedom	MS	F	p
TECHN	26092846	4	6523212.00	18943.38	0.00
Error	1396351	4055	344.35		

(b) print_tokens2

	SS	Degrees of freedom	MS	F	p
TECHN	7715385	4	1928846.00	2228.324	0.00
Error	2414170	2789	865.60		

(c) schedule

	SS	Degrees of freedom	MS	F	p
TECHN	12773988	4	3193497.00	12569.990	0.00
Error	672998	2649	254.06		

(d) schedule2

	SS	Degrees of freedom	MS	F	p
TECHN	10211810	4	2552953.00	7735.107	0.00
Error	521145	1579	330.05		

(e) tcas

	SS	Degrees of freedom	MS	F	p
TECHN	22010962	4	5502741.00	12992.020	0.00
Error	1512913	3572	423.55		

(f) tot_info

	SS	Degrees of freedom	MS	F	p
TECHN	14603466	4	3650866.00	8159.339	0.00
Error	1111457	2484	447.45		

(g) replace

	SS	Degrees of freedom	MS	F	p
TECHN	48633443	4	12158361.00	335474	0.00
Error	199695	5510	36.24		

(h) space

TABLE B.1: ANOVA Analyses, Statement Level Techniques, Individual Programs

Grouping	Means	Techniques
A	81.246	st-cov-fb
A	80.923	st-cov-nofb
A	80.882	st-fep-fb
A	80.032	st-fep-nofb

(a) print_tokens

Grouping	Means	Techniques
A	82.147	st-fep-fb
B	79.701	st-cov-nofb
B	79.615	st-fep-nofb
B	79.201	st-cov-fb

(b) print_tokens2

Grouping	Means	Techniques
A	57.553	st-fep-fb
A	55.798	st-cov-fb
B	48.187	st-cov-nofb
B	48.150	st-fep-nofb

(c) schedule

Grouping	Means	Techniques
A	71.775	st-fep-fb
A	70.005	st-cov-nofb
A	69.498	st-fep-nofb
B	66.410	st-cov-fb

(d) schedule2

Grouping	Means	Techniques
A	84.740	st-fep-nofb
A	83.667	st-cov-nofb
A	82.191	st-fep-fb
B	70.692	st-cov-fb

(e) tcas

Grouping	Means	Techniques
A	80.962	st-fep-nofb
A	80.584	st-fep-fb
B	77.382	st-cov-nofb
B	74.606	st-cov-fb

(f) tot_info

Grouping	Means	Techniques
A	80.822	st-fep-nofb
A	80.627	st-fep-fb
A	78.631	st-cov-nofb
B	64.910	st-cov-fb

(g) replace

Grouping	Means	Techniques
A	95.343	st-fep-fb
B	93.646	st-fep-nofb
B	93.583	st-cov-nofb
B	93.072	st-cov-fb

(h) space

TABLE B.2: Bonferroni Analyses, Statement Level Techniques, Individual Programs

	SS	Degrees of freedom	MS	F	p
TECHN	19796189	4	4949047.00	8378.731	0.00
Error	1971059	3337	590.67		

(a) print_tokens

	SS	Degrees of freedom	MS	F	p
TECHN	22457946	4	5614487.00	16252.620	0.00
Error	1379732	3994	345.45		

(b) print_tokens2

	SS	Degrees of freedom	MS	F	p
TECHN	6272983	4	1568246.00	1499.704	0.00
Error	2960386	2831	1045.70		

(c) schedule

	SS	Degrees of freedom	MS	F	p
TECHN	11780218	4	2945055.00	12326.930	0.00
Error	620933	2599	238.91		

(d) schedule2

	SS	Degrees of freedom	MS	F	p
TECHN	8493174	4	2123294.00	7107.707	0.00
Error	388649	1301	298.73		

(e) tcas

	SS	Degrees of freedom	MS	F	p
TECHN	21283035	4	5320759.00	13091.780	0.00
Error	1429785	3518	406.42		

(f) tot_info

	SS	Degrees of freedom	MS	F	p
TECHN	15958536	4	3989634.00	10152.800	0.00
Error	975717	2484	392.96		

(g) replace

	SS	Degrees of freedom	MS	F	p
TECHN	46987341	4	11746835.00	187251.800	0.00
Error	345407	5506	62.73		

(h) space

TABLE B.3: ANOVA Analyses, Basic Function Level Techniques, Individual Programs

Grouping	Means	Techniques
A	78.615	fn-fep-nofb
A	77.803	fn-fep-fb
A B	77.142	fn-cov-nofb
B	74.160	fn-cov-fb

(a) print_tokens

Grouping	Means	Techniques
A	80.447	fn-fep-fb
B	77.538	fn-fep-nofb
C	75.243	fn-cov-nofb
D	65.627	fn-cov-fb

(b) print_tokens2

Grouping	Means	Techniques
A	48.707	fn-fep-fb
A	47.988	fn-fep-nofb
A	46.893	fn-cov-nofb
A	44.471	fn-cov-fb

(c) schedule

Grouping	Means	Techniques
A	72.077	fn-cov-nofb
B	68.664	fn-fep-nofb
B	67.402	fn-fep-fb
C	60.546	fn-cov-fb

(d) schedule2

Grouping	Means	Techniques
A	84.029	fn-cov-nofb
B	80.400	fn-cov-fb
B	79.763	fn-fep-fb
B	77.404	fn-fep-nofb

(e) tcas

Grouping	Means	Techniques
A	79.297	fn-fep-fb
A	79.254	fn-fep-nofb
A B	77.241	fn-cov-fb
B	74.989	fn-cov-nofb

(f) tot_info

Grouping	Means	Techniques
A	81.859	fn-fep-fb
A	80.800	fn-cov-nofb
A	80.793	fn-fep-nofb
B	76.890	fn-cov-fb

(g) replace

Grouping	Means	Techniques
A	94.119	fn-cov-fb
B	93.175	fn-cov-nofb
C	91.030	fn-fep-nofb
C	91.010	fn-fep-fb

(h) space

TABLE B.4: Bonferroni Analyses, Basic Function Level Techniques, Individual Programs

	SS	Degrees of freedom	MS	F	p
TECHN	41663772	8	5207971.00	9583.498	0.00
Error	3632837	6685	543.43		

(a) print_tokens

	SS	Degrees of freedom	MS	F	p
TECHN	48550792	8	6068849.00	17596.070	0.00
Error	2776083	8049	344.90		

(b) print_tokens2

	SS	Degrees of freedom	MS	F	p
TECHN	13988368	8	1748546.00	1828.398	0.00
Error	5374556	5620	956.33		

(c) schedule

	SS	Degrees of freedom	MS	F	p
TECHN	24554206	8	3069276.00	12448.550	0.00
Error	1293930	5248	246.56		

(d) schedule2

	SS	Degrees of freedom	MS	F	p
TECHN	18704984	8	2338123.00	7401.448	0.00
Error	909794	2880	315.90		

(e) tcas

	SS	Degrees of freedom	MS	F	p
TECHN	43293998	8	5411750.00	13038.820	0.00
Error	2942697	7090	415.05		

(f) tot_info

	SS	Degrees of freedom	MS	F	p
TECHN	30562002	8	3820250.00	9091.328	0.00
Error	2087174	4967.00	420.21		

(g) replace

	SS	Degrees of freedom	MS	F	p
TECHN	95620784	8	11952598.00	241550.700	0.00
Error	545102	11016	49.49		

(h) space

TABLE B.5: ANOVA Analyses, Function Versus Statement Level Techniques, Individual Programs

Grouping	Means	Techniques
A	81.246	st-cov-fb
A	80.923	st-cov-nofb
A	80.882	st-fep-fb
A B	80.032	st-fep-nofb
A B	78.615	fn-fep-nofb
A B	77.803	fn-fep-fb
B C	77.142	fn-cov-nofb
C	74.160	fn-cov-fb

(a) print_tokens

Grouping	Means	Techniques
A	82.147	st-fep-fb
A B	80.447	fn-fep-fb
A B C	79.701	st-cov-nofb
A B C	79.615	st-fep-nofb
B C	79.201	st-cov-fb
C D	77.538	fn-fep-nofb
D	75.243	fn-cov-nofb
E	65.627	fn-cov-fb

(b) print_tokens2

Grouping	Means	Techniques
A	57.553	st-fep-fb
A	55.798	st-cov-fb
B	48.707	fn-fep-fb
B	48.187	st-cov-nofb
B	48.150	st-fep-nofb
B	47.988	fn-fep-nofb
B	46.893	fn-cov-nofb
B	44.471	fn-cov-fb

(c) schedule

Grouping	Means	Techniques
A	72.077	fn-cov-nofb
A	71.775	st-fep-fb
A B	70.005	st-cov-nofb
B	69.498	st-fep-nofb
C	68.664	fn-fep-nofb
C	67.402	fn-fep-fb
C	66.410	st-cov-fb
D	60.546	fn-cov-fb

(d) schedule2

Grouping	Means	Techniques
A	84.740	st-fep-nofb
A B	84.029	fn-cov-nofb
A B	83.667	st-cov-nofb
A B C	82.191	st-fep-fb
B C	80.400	fn-cov-fb
B C	79.763	fn-fep-fb
C	77.404	fn-fep-nofb
D	70.692	st-cov-fb

(e) tcas

Grouping	Means	Techniques
A	80.962	st-fep-nofb
A	80.584	st-fep-fb
A B	79.297	fn-fep-fb
A B	79.254	fn-fep-nofb
B C	77.382	st-cov-nofb
B C	77.241	fn-cov-fb
C	74.989	fn-cov-nofb
C	74.606	st-cov-fb

(f) tot_info

Grouping	Means	Techniques
A	81.859	fn-fep-fb
A B	80.822	st-fep-nofb
A B	80.800	fn-cov-nofb
A B	80.793	fn-fep-nofb
A B	80.627	st-fep-fb
A B	78.631	st-cov-nofb
B	76.890	fn-cov-fb
C	64.910	st-cov-fb

(g) replace

Grouping	Means	Techniques
A	95.343	st-fep-fb
B	94.119	fn-cov-fb
B C	93.646	st-fep-nofb
B C	93.583	st-cov-nofb
C	93.175	fn-cov-nofb
C	93.072	st-cov-fb
D	91.030	fn-fep-nofb
D	91.010	fn-fep-fb

(h) space

TABLE B.6: Bonferroni Analyses, Function Versus Statement Level Techniques, Individual Programs

	SS	Degrees of freedom	MS	F	p
TECHN	57233358	12	4769447.00	8727.343	0.00
Error	5437076	9949	546.49		

(a) print_tokens

	SS	Degrees of freedom	MS	F	p
TECHN	69334256	12	5777855.00	17780.040	0.00
Error	3935628	12111	324.96		

(b) print_tokens2

	SS	Degrees of freedom	MS	F	p
TECHN	20962038	12	1746837.00	1800.094	0.00
Error	8256282	8508	970.41		

(c) schedule

	SS	Degrees of freedom	MS	F	p
TECHN	35476722	12	2956394.00	11978.550	0.00
Error	1926577	7806	246.81		

(d) schedule2

	SS	Degrees of freedom	MS	F	p
TECHN	24831657	12	2069305.00	7265.014	0.00
Error	1162682	4082	284.83		

(e) tcas

	SS	Degrees of freedom	MS	F	p
TECHN	64353218	12	5362768.00	13095.840	0.00
Error	4365287	10660	409.50		

(f) tot_info

	SS	Degrees of freedom	MS	F	p
TECHN	48586487	12	4048874.00	11644.180	0.00
Error	2608221	7501	347.72		

(g) replace

	SS	Degrees of freedom	MS	F	p
TECHN	144917720	12	1207647700.	232624.200	0.00
Error	859957	16565	51.914		

(h) space

TABLE B.7: ANOVA Analyses, All Function Level Techniques, Individual Programs

Grouping	Means	Techniques
A	78.894	fn-fi-fep-nofb
A	78.615	fn-fep-nofb
A B	77.997	fn-diff-fep-nofb
A B	77.886	fn-fi-fep-fb
A B	77.872	fn-diff-fep-fb
A B	77.803	fn-fep-fb
A B	77.704	fn-fi-cov-nofb
A B	77.142	fn-cov-nofb
B	75.346	fn-fi-cov-fb
B	74.422	fn-diff-cov-nofb
B	74.160	fn-cov-fb
C	58.945	fn-diff-cov-fb

(a) print_tokens

Grouping	Means	Techniques
A	85.015	fn-diff-fep-fb
A	85.008	fn-diff-fep-nofb
B	81.083	fn-fi-fep-nofb
B	80.447	fn-fep-fb
B C	79.538	fn-fi-fep-fb
C D	77.538	fn-fep-nofb
D E	75.704	fn-diff-cov-nofb
D E	75.243	fn-cov-nofb
E	73.735	fn-fi-cov-nofb
F	68.119	fn-fi-cov-fb
F	65.627	fn-cov-fb
G	52.108	fn-diff-cov-fb

(b) print_tokens2

Grouping	Means	Techniques
A	61.249	fn-diff-cov-nofb
B	51.147	fn-diff-cov-fb
B	50.610	fn-diff-fep-nofb
B	50.558	fn-diff-fep-fb
B	49.504	fn-fi-fep-nofb
B C	48.727	fn-fi-fep-fb
B C	48.707	fn-fep-fb
B C	48.463	fn-fi-cov-nofb
B C	47.988	fn-fep-nofb
B C	46.893	fn-cov-nofb
C	44.950	fn-fi-cov-fb
C	44.471	fn-cov-fb

(c) schedule

Grouping	Means	Techniques
A	72.319	fn-diff-cov-nofb
A B	72.077	fn-cov-nofb
A B	71.687	fn-fi-cov-nofb
A B C	70.199	fn-diff-fep-nofb
A B C	70.078	fn-diff-fep-fb
B C	69.288	fn-fi-fep-nofb
B C	69.239	fn-fi-fep-fb
C	68.664	fn-fep-nofb
C	67.402	fn-fep-fb
D	60.546	fn-cov-fb
D	60.522	fn-fi-cov-fb
E	54.325	fn-diff-cov-fb

(d) schedule2

Grouping	Means	Techniques
A	84.029	fn-cov-nofb
A	84.029	fn-fi-cov-nofb
A B	82.016	fn-fi-fep-nofb
A B	80.400	fn-cov-fb
A B	80.400	fn-fi-cov-fb
A B	79.763	fn-fep-fb
B	78.926	fn-fi-fep-fb
B	77.785	fn-diff-fep-nofb
B	77.726	fn-diff-fep-fb
B	77.404	fn-fep-nofb
C	70.063	fn-diff-cov-nofb
D	56.299	fn-diff-cov-fb

(e) tcas

Grouping	Means	Techniques
A	81.288	fn-diff-fep-fb
A	81.263	fn-diff-fep-nofb
A B	79.517	fn-fi-fep-fb
A B	79.426	fn-fi-fep-nofb
A B	79.297	fn-fep-fb
A B	79.254	fn-fep-nofb
B C	77.326	fn-fi-cov-fb
B C	77.241	fn-cov-fb
C D	75.078	fn-fi-cov-nofb
C D	74.989	fn-cov-nofb
D	73.461	fn-diff-cov-nofb
D	72.370	fn-diff-cov-fb

(f) tot_info

TABLE B.8: Bonferroni Analyses, All Function Level Techniques, Individual Programs, First Six Programs

Grouping			Means	Techniques
A			83.734	fn-fi-fep-nofb
A			83.559	fn-diff-fep-nofb
A			82.986	fn-fi-fep-fb
A			82.791	fn-diff-fep-fb
A	B		81.859	fn-fep-fb
A	B		80.800	fn-cov-nofb
A	B		80.793	fn-fep-nofb
A	B		80.700	fn-fi-cov-nofb
	B	C	78.488	fn-fi-cov-fb
		C	76.890	fn-cov-fb
		C	76.421	fn-diff-cov-nofb
		C	74.928	fn-diff-cov-fb

(a) replace

Grouping					Means	Techniques
A					96.142	fn-fi-cov-fb
B					94.959	fn-fi-cov-nofb
B	C				94.829	fn-diff-cov-fb
B	C	D			94.119	fn-cov-fb
	C	D	E		93.991	fn-diff-cov-nofb
		D	E	F	93.600	fn-fi-fep-fb
		D	E	F	93.462	fn-diff-fep-fb
			E	F	93.175	fn-cov-nofb
				F	92.795	fn-diff-fep-nofb
				F	92.713	fn-fi-fep-nofb
G					91.030	fn-fep-nofb
G					91.010	fn-fep-fb

(b) space

TABLE B.9: Bonferroni Analyses, All Function Level Techniques, Individual Programs, Last Two Programs

CHAPTER C

TUKEY TABLES FOR STUDIES OF TECHNIQUES INCORPORATING TEST COST AND FAULT SEVERITY ESTIMATIONS

Tables C.1, C.2, and C.3 present the results of applying Tukey tests to each (test-cost distribution, prioritization technique) pair compared with each other (test-cost distribution, prioritization technique) pair. (Given sufficient space, the three tables would be concatenated left-to-right, yielding a single table; for readability we have partitioned that single table into three.) Entries shown in italics are statistically significant ($\alpha = .05$).

	unit ac-f	unit ac-s	unit fi-ac-f	unit rand	random ac-f	random ac-s
unit ac-f		0.997735	*0.003449*	*0.000043*	0.393361	0.764031
unit ac-s	0.997735		*0.000045*	*0.000043*	*0.00514*	*0.032925*
unit fi-ac-f	*0.003449*	*0.000045*		*0.000043*	0.994839	0.904847
unit rand	*0.000043*	*0.000043*	*0.000043*		*0.000043*	*0.000043*
random ac-f	0.393361	*0.00514*	0.994839	*0.000043*		1.000000
random ac-s	0.764031	*0.032925*	0.904847	*0.000043*	1.000000	
random fi-ac-f	*0.014002*	*0.000059*	1.000000	*0.000043*	0.999854	0.985423
random rand	*0.000043*	*0.000043*	*0.000043*	1.000000	*0.000043*	*0.000043*
normal ac-f	0.999996	0.712642	0.096081	*0.000043*	0.957261	0.998885
normal ac-s	1.000000	0.999785	*0.001313*	*0.000043*	0.248442	0.599819
normal fi-ac-f	*0.000383*	*0.000043*	1.000000	*0.000043*	0.901452	0.588792
normal rand	*0.000043*	*0.000043*	*0.000043*	1.000000	*0.000043*	*0.000043*
Mozilla ac-f	0.829598	*0.047049*	0.856514	*0.000043*	1.000000	1.000000
Mozilla ac-s	0.999978	0.63592	0.128444	*0.000043*	0.976088	0.999629
Mozilla fi-ac-f	*0.016979*	*0.000065*	1.000000	*0.000043*	0.999926	0.989813
Mozilla rand	*0.000043*	*0.000043*	*0.000043*	1.000000	*0.000043*	*0.000043*
QTB ac-f	0.432404	*0.00639*	0.992209	*0.000043*	1.000000	1.000000
QTB ac-s	0.999164	0.413233	0.262674	*0.000043*	0.997111	0.999993
QTB fi-ac-f	*0.012322*	*0.000056*	1.000000	*0.000043*	0.999779	0.981813
QTB rand	*0.000043*	*0.000043*	*0.000043*	1.000000	*0.000043*	*0.000043*

TABLE C.1: Results of Tukey Tests – Table 1

	random fi-ac-f	random rand	normal ac-f	normal ac-s	normal fi-ac-f	normal rand	Mozilla ac-f
unit ac-f	*0.014002*	*0.000043*	0.999996	1.000000	*0.000383*	*0.000043*	0.829598
unit ac-s	*0.000059*	*0.000043*	0.712642	0.999785	*0.000043*	*0.000043*	*0.047049*
unit fi-ac-f	1.000000	*0.000043*	0.096081	*0.001313*	1.000000	*0.000043*	0.856514
unit rand	*0.000043*	1.000000	*0.000043*	*0.000043*	*0.000043*	1.000000	*0.000043*
random ac-f	0.999854	*0.000043*	0.957261	0.248442	0.901452	*0.000043*	1.000000
random ac-s	0.985423	*0.000043*	0.998885	0.599819	0.588792	*0.000043*	1.000000
random fi-ac-f		*0.000043*	0.240072	*0.005854*	0.999999	*0.000043*	0.971641
random rand	*0.000043*		*0.000043*	*0.000043*	*0.000043*	1.000000	*0.000043*
normal ac-f	0.240072	*0.000043*		0.99988	*0.018429*	*0.000043*	0.999644
normal ac-s	*0.005854*	*0.000043*	0.99988		*0.000152*	*0.000043*	0.681292
normal fi-ac-f	0.999999	*0.000043*	*0.018429*	*0.000152*		*0.000043*	0.505272
normal rand	*0.000043*	1.000000	*0.000043*	*0.000043*	*0.000043*		*0.000043*
Mozilla ac-f	0.971641	*0.000043*	0.999644	0.681292	0.505272	*0.000043*	
Mozilla ac-s	0.300982	*0.000043*	1.000000	0.999584	*0.026659*	*0.000043*	0.999899
Mozilla fi-ac-f	1.000000	*0.000043*	0.269666	*0.0072*	0.999997	*0.000043*	0.979232
Mozilla rand	*0.000043*	1.000000	*0.000043*	*0.000043*	*0.000043*	1.000000	*0.000043*
QTB ac-f	0.999721	*0.000043*	0.968	0.279434	0.878459	*0.000043*	1.000000
QTB ac-s	0.508514	*0.000043*	1.000000	0.993719	0.069097	*0.000043*	0.999999
QTB fi-ac-f	1.000000	*0.000043*	0.221922	*0.005086*	1.000000	*0.000043*	0.965632
QTB rand	*0.000043*	1.000000	*0.000043*	*0.000043*	*0.000043*	1.000000	*0.000043*

TABLE C.2: Results of Tukey Tests – Table 2

	Mozilla ac-s	Mozilla fi-ac-f	Mozilla rand	QTB ac-f	QTB ac-s	QTB fi-ac-f	QTB rand
unit ac-f	0.999978	*0.016979*	*0.000043*	0.432404	0.999164	*0.012322*	*0.000043*
unit ac-s	0.63592	*0.000065*	*0.000043*	*0.00639*	0.413233	*0.000056*	*0.000043*
unit fi-ac-f	0.128444	1.000000	*0.000043*	0.992209	0.262674	1.000000	*0.000043*
unit rand	*0.000043*	*0.000043*	1.000000	*0.000043*	*0.000043*	*0.000043*	1.000000
random ac-f	0.976088	0.999926	*0.000043*	1.000000	0.997111	0.999779	*0.000043*
random ac-s	0.999629	0.989813	*0.000043*	1.000000	0.999993	0.981813	*0.000043*
random fi-ac-f	0.300982	1.000000	*0.000043*	0.999721	0.508514	1.000000	*0.000043*
random rand	*0.000043*	*0.000043*	1.000000	*0.000043*	*0.000043*	*0.000043*	1.000000
normal ac-f	1.000000	0.269666	*0.000043*	0.968	1.000000	0.221922	*0.000043*
normal ac-s	0.999584	*0.0072*	*0.000043*	0.279434	0.993719	*0.005086*	*0.000043*
normal fi-ac-f	*0.026659*	0.999997	*0.000043*	0.878459	0.069097	1.000000	*0.000043*
normal rand	*0.000043*	*0.000043*	1.000000	*0.000043*	*0.000043*	*0.000043*	1.000000
Mozilla ac-f	0.999899	0.979232	*0.000043*	1.000000	0.999999	0.965632	*0.000043*
Mozilla ac-s		0.334697	*0.000043*	0.982841	1.000000	0.280043	*0.000043*
Mozilla fi-ac-f	0.334697		*0.000043*	0.999852	0.549053	1.000000	*0.000043*
Mozilla rand	*0.000043*	*0.000043*		*0.000043*	*0.000043*	*0.000043*	1.000000
QTB ac-f	0.982841	0.999852	*0.000043*		0.998215	0.999588	*0.000043*
QTB ac-s	1.000000	0.549053	*0.000043*	0.998215		0.482269	*0.000043*
QTB fi-ac-f	0.280043	1.000000	*0.000043*	0.999588	0.482269		*0.000043*
QTB rand	*0.000043*	*0.000043*	1.000000	*0.000043*	*0.000043*	*0.000043*	

TABLE C.3: Results of Tukey Tests – Table 3