Prioritizing Test Cases For Regression Testing

Gregg Rothermel, *Member*, *IEEE Computer Society*,
Roland H. Untch, *Member*, *IEEE Computer Society*, Chengyun Chu, and
Mary Jean Harrold, *Member*, *IEEE Computer Society*

Abstract—Test case prioritization techniques schedule test cases for execution in an order that attempts to increase their effectiveness at meeting some performance goal. Various goals are possible; one involves *rate of fault detection*—a measure of how quickly faults are detected within the testing process. An improved rate of fault detection during testing can provide faster feedback on the system under test and let software engineers begin correcting faults earlier than might otherwise be possible. One application of prioritization techniques involves regression testing—the retesting of software following modifications; in this context, prioritization techniques can take advantage of information gathered about the previous execution of test cases to obtain test case orderings. In this paper, we describe several techniques for using test execution information to prioritize test cases for regression testing, including: 1) techniques that order test cases based on their total coverage of code components, 2) techniques that order test cases based on their coverage of code components not previously covered, and 3) techniques that order test cases based on their estimated ability to reveal faults in the code components that they cover. We report the results of several experiments in which we applied these techniques to various test suites for various programs and measured the rates of fault detection achieved by the prioritized test suites, comparing those rates to the rates achieved by untreated, randomly ordered, and optimally ordered suites. Analysis of the data shows that each of the prioritization techniques studied improved the rate of fault detection of test suites, and this improvement occurred even with the least expensive of those techniques. The data also shows, however, that considerable room remains for improvement. The studies highlight several cost-benefit trade-offs among the techniques studied, as well as several opportunities for future work.

Index Terms—Test case prioritization, regression testing, software testing, empirical studies.

1 Introduction

Software engineers often save the test suites they develop for their software so that they can reuse those test suites later as the software evolves. Such test suite reuse, in the form of regression testing, is pervasive in the software industry [24] and, together with other regression testing activities, has been estimated to account for as much as one-half of the cost of software maintenance [4], [20]. Running all of the test cases in a test suite, however, can require a large amount of effort. For example, one of our industrial collaborators reports that for one of its products of about 20,000 lines of code, the entire test suite requires seven weeks to run.

For this reason, researchers have considered various techniques for reducing the cost of regression testing, including regression test selection, and test suite minimization techniques. Regression test selection techniques (e.g., [5], [7], [21], [29]) reduce the cost of regression testing by selecting an appropriate subset of the existing test suite based on information about the program, modified version, and test suite. Test suite minimization techniques (e.g., [6], [15], [30], [37]) lower costs by reducing a test suite to a minimal subset that maintains equivalent coverage of the original test suite with respect to a particular test adequacy criterion.

Regression test selection and test suite minimization techniques, however, can have drawbacks. For example, although some empirical evidence indicates that, in certain cases, there is little or no loss in the ability of a minimized test suite to reveal faults in comparison to its unminimized original [37], [38], other empirical evidence shows that the fault detection capabilities of test suites can be severely compromised by minimization [30]. Similarly, although there are *safe* regression test selection techniques (e.g., [3], [7], [29], [34]) that can ensure that the selected subset of a test suite has the same fault detection capabilities as the original test suite, the conditions under which safety can be achieved do not always hold [28], [29].

Test case prioritization techniques [31], [36] provide another method for assisting with regression testing.[1] These techniques let testers order their test cases so that those test cases with the highest priority, according to some criterion, are executed earlier in the regression testing process than lower priority test cases. For example, testers might wish to schedule test cases in an order that achieves code coverage at the fastest rate possible, exercises features in order of expected frequency of use, or exercises

- *G. Rothermel is with the Department of Computer Science, Oregon State University, Corvallis, OR. E-mail: grother@cs.orst.edu.*
- *R.H. Untch is with the Department of Computer Science, Middle Tennessee State University, Murfreesboro, TN. E-mail: untch@mtsu.edu.*
- *C. Chu is with Microsoft, Inc., One Microsoft Way, Redmond, WA 98052-6399. E-mail: chchu@microsoft.com.*
- *M.J. Harrold is with the College of Computing, Georgia Institute of Technology, 801 Atlantic Dr., Atlanta, GA. E-mail: harrold@cc.gatech.edu.*

Manuscript received 23 Dec. 1999; accepted 21 Aug. 2000.
Recommended for acceptance by A.A. Andrews.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 111129.

1. Some test case prioritization techniques may be applicable during the initial testing of software [1]. In this paper, however, we are concerned only with regression testing. Section 2 discusses other applications of prioritization and related work on prioritization in further detail.

0098-5589/01/$10.00 © 2001 IEEE

subsystems in an order that reflects their historically demonstrated propensity to fail.

When the time required to reexecute an entire test suite is short, test case prioritization may not be cost-effective—it may be sufficient simply to schedule test cases in any order. When the time required to execute an entire test suite is sufficiently long, however, test-case prioritization may be beneficial because, in this case, meeting testing goals earlier can yield meaningful benefits.

Because test case prioritization techniques do not themselves discard test cases, they can avoid the drawbacks that can occur when regression test selection and test suite minimization discard test cases. Alternatively, in cases where the discarding of test cases is acceptable, test case prioritization can be used in conjunction with regression test selection or test suite minimization techniques to prioritize the test cases in the selected or minimized test suite. Further, test case prioritization can increase the likelihood that, if regression testing activities are unexpectedly terminated, testing time will have been spent more beneficially than if test cases were not prioritized.

In this paper, we describe several techniques for prioritizing test cases for regression testing. We then describe several empirical studies we performed with these techniques to evaluate their ability to improve *rate of fault detection*—a measure of how quickly faults are detected within the testing process. An improved rate of fault detection during regression testing provides earlier feedback on a system under test and lets debugging activities begin earlier than might otherwise be possible. Our results indicate that test case prioritization can significantly improve the rate of fault detection of test suites. Our results also highlight several cost-benefit trade-offs between various techniques.

The next section of this paper precisely describes the test case prioritization problem, presents several prioritization techniques, and discusses previous work on prioritization. Section 3 presents the design, results, and analysis of our empirical studies. Section 4 discusses our results and their practical implications, and Section 5 presents overall conclusions and discusses future work.

2 Test Case Prioritization

We formally define the test case prioritization problem as follows:

Definition 1. *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 represents the set of all possible prioritizations (orderings) of T, and f is a function that, applied to any such ordering, yields an *award value* for that ordering. (For simplicity, and without loss of generality, the definition assumes that higher award values are preferable to lower ones.)

There are several aspects of the test case prioritization problem that are worth describing further. First, there are many possible goals of prioritization, including the following:

- Testers may wish to increase the rate of fault detection of a test suite—that is, the likelihood of revealing faults earlier in a run of regression tests using that test suite.
- Testers may wish to increase the coverage of coverable code in the system under test at a faster rate, thus allowing a code coverage criterion to be met earlier in the test process.
- Testers may wish to increase their confidence in the reliability of the system under test at a faster rate.
- Testers may wish to increase the rate at which high-risk faults are detected by a test suite, thus locating such faults earlier in the testing process.
- Testers may wish to increase the likelihood of revealing faults related to specific code changes earlier in the regression testing process.

Here, these goals are stated qualitatively. To measure the success of a prioritization technique in meeting any such goal, however, we must describe the goal quantitatively. In Definition 1, f represents such a quantification. Later in this paper, we will precisely define one particular function f for use in quantifying the first of these goals.

Second, depending upon the choice of f, the test case prioritization problem may be intractable or undecidable. For example, given a function f that quantifies whether a test suite achieves statement coverage at the fastest rate possible, an efficient solution to the test case prioritization problem would provide an efficient solution to the knapsack problem.[2] Similarly, given a function f that quantifies whether a test suite detects faults at the fastest rate possible, a precise solution to the test case prioritization problem would provide a solution to the halting problem. In such cases, prioritization techniques must be heuristics.

Third, test case prioritization can be used either in the initial testing of software or in the regression testing of software. One difference between these two applications is that, in the case of regression testing, prioritization techniques can use information gathered in previous runs of existing test cases to help prioritize the test cases for subsequent runs.

Fourth, it is useful to distinguish two varieties of test case prioritization: general test case prioritization and version-specific test case prioritization. In *general test case prioritization*, given program P and test suite T, we prioritize the test cases in T with the intent of finding an ordering of test cases that will be useful over a succession of subsequent modified versions of P. Thus, general test case prioritization can be performed following the release of some version of the program during off-peak hours, and the cost of performing the prioritization is amortized over the subsequent releases. It is hoped that the resulting prioritized suite will be more successful than the original suite at meeting the goal of the prioritization, *on average* over those subsequent releases.

2. Informally, the knapsack problem is the problem of, given a set U whose elements each have a cost and a value, and given a size constraint and a value goal, finding a subset U' of U such that U' meets the given size constraint and the given value goal. For a more formal treatment see [11].

TABLE 1
A Catalog of Prioritization Techniques

Code	*Mnemonic*	*Description*
M_1	untreated	no prioritization
M_2	random	randomized ordering
M_3	optimal	ordered to optimize rate of fault detection
M_4	stmt-total	prioritize in order of coverage of statements
M_5	stmt-addtl	prioritize in order of coverage of statements not yet covered
M_6	branch-total	prioritize in order of coverage of branches
M_7	branch-addtl	prioritize in order of coverage of branches not yet covered
M_8	FEP-total	prioritize in order of total probability of exposing faults
M_9	FEP-addtl	prioritize in order of total probability of exposing faults, adjusted to consider effects of previous test cases

In contrast, in *version-specific test case prioritization*, given program P and test suite T, we prioritize the test cases in T with the intent of finding an ordering that will be useful on a specific version P' of P. Version-specific prioritization is performed after a set of changes have been made to P and prior to regression testing P'. Because this prioritization is accomplished after P' is available, care must be taken to keep the cost of performing the prioritization from excessively delaying the very regression testing activities it is intended to facilitate. The prioritized test suite may be more effective at meeting the goal of the prioritization for P' in particular than would a test suite resulting from general test case prioritization, but may be less effective on average over a succession of subsequent releases.

Typically—though not necessarily—general test case prioritization does not use information about specific modified versions of P, whereas version-specific prioritization does use such information. Of course, it is possible for general test case prioritization techniques to incorporate information about expected modifications to improve the average performance of prioritized test suites over a succession of program versions, and it is possible to use prioritization techniques that ignore the modified program as version-specific techniques.

Fifth, it is also possible to integrate test case prioritization with regression test selection or test suite minimization techniques—for example, by prioritizing a test suite selected by a regression test selection algorithm, or by prioritizing the minimal test suite returned by a test suite minimization algorithm.

Finally, given any prioritization goal, various *prioritization techniques* may be applied to a test suite with the aim of meeting that goal. For example, in an attempt to increase the rate of fault detection of test suites, we might prioritize test cases in terms of the extent to which they execute modules that, measured historically, have tended to fail. Alternatively, we might prioritize test cases 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 ordering of test cases.

In this paper, we restrict our attention, focusing on general test case prioritization in application to regression testing, independent of regression test selection and test suite minimization. We focus on a specific goal and function f, and we evaluate the abilities of several prioritization techniques to help us meet this goal.

2.1 Prioritization for Rate of Fault Detection

Our focus in this paper is the first goal listed at the beginning of Section 2: increasing the likelihood of revealing faults earlier in the testing process. Informally, we describe this goal as one of improving our test suite's *rate of fault detection*: We describe a function f that quantifies this goal in Section 3.2.

As we suggested in Section 1, there are several motivations for meeting this goal. An improved rate of fault detection during regression testing can let software engineers begin their debugging activities earlier than might otherwise be possible, speeding the release of the software. An improved rate of fault detection can also provide faster feedback on the system under test and provide earlier evidence when quality goals have not been met, thus allowing strategic decisions about release schedules to be made earlier than might otherwise be possible. Further, in a testing situation in which the amount of testing time that will be available is uncertain (for example, when market pressures may force a release of the product prior to execution of all test cases), such prioritization can increase the likelihood that, whenever the testing process is terminated, testing resources will have been spent more cost-effectively in relation to potential fault detection than they might otherwise have been.

In this paper, we consider nine different test case prioritization techniques (see Table 1). The first three techniques serve as experimental controls (though not actually "techniques" in a practical sense, we refer to them as such to simplify the presentation.) The last six techniques represent heuristics that could be implemented using software tools; all of these techniques use test coverage information, produced by prior executions of test cases, to prioritize test cases for subsequent execution. A source of motivation for such approaches is the conjecture that the availability of test execution data can be an asset; however, such approaches also make the assumption that past test execution data can be used to predict, with sufficient

accuracy, subsequent execution behavior. In practice, code modifications made to create a new version may alter test execution patterns; an issue impacting the efficacy of test case prioritization techniques is whether these alterations will significantly impact the predictive value of past execution data.

We next describe the nine techniques listed in Table 1 in turn.

M_1: **No prioritization**. To facilitate our empirical studies, one prioritization technique that we consider is simply the application of no technique; this lets us consider "untreated" test suites and serves as a control.

M_2: **Random prioritization**. The success of an untreated test suite in meeting a goal may depend upon the manner in which the test suite is initially constructed. Therefore, as an additional control in our studies, we apply random prioritization, in which we randomly order the test cases in a test suite.

M_3: **Optimal prioritization**. As we shall discuss in Section 3, to measure the effects of prioritization techniques on the rate of fault detection, our empirical studies use programs that contain known faults. Given program P and a set of known faults for P, if we can determine, for test suite T, which test cases in T expose which faults in P, then we can determine an optimal ordering of the test cases in T for maximizing T's rate of fault detection for that set of faults. In practice, of course, this is not a practical technique, as it requires a priori knowledge of the existence of faults and of which test cases expose which faults; however, by using this technique in our empirical studies, we can gain insight into the success of other practical heuristics, by comparing their solutions to optimal solutions.

An algorithm that always determines an optimal test-case ordering may have to consider all possible test-case orderings and, therefore, must have a worst-case runtime exponential in test suite size. Many of the test suites we use in our empirical studies are too large to support the practical use of such an algorithm; thus, in our empirical studies, we have employed a greedy "optimal" prioritization algorithm. Given a program P with a set of faulty versions, a test suite T, and information on which test cases in T expose which faults, our algorithm iteratively selects the test case in T that exposes the most faults not yet exposed by a selected test case, until test cases that expose all faults have been selected. When test cases that expose all faults have been selected by this algorithm, remaining test cases must be prioritized by some method. Given the measure of rate of fault detection that we employ in this work, however, this ordering of subsequent test cases has no effect on rate of fault detection (this shall become clear following discussion of our effectiveness measure in Section 3.2). Thus, our algorithm prioritizes these remaining test cases in order of their appearance in the original test suite.

This greedy prioritization algorithm may not always choose the optimal test case ordering. To see this, suppose a program contains four faults, and suppose our test suite for that program contains three test cases that detect those faults as shown in Table 2. Our greedy algorithm may select test case t_1 first, test case t_2 second, and test case t_3 third. However, the optimal test case orderings in this case are t_2, t_3, t_1 and t_3, t_2, t_1. Despite this fact, as we shall show, our algorithm provides a useful benchmark against which to measure practical techniques because we know that an optimal ordering could perform no worse than the ordering that we calculate. For brevity, in the rest of this paper, we refer to our technique that incorporates this algorithm as *optimal prioritization*.

TABLE 2
A Case in Which the Greedy "Optimal" Prioritization Algorithm May Not Produce an Optimal Solution

Test Case	Fault			
	1	2	3	4
t_1	X	X		
t_2	X			X
t_3		X	X	

M_4: **Total statement coverage prioritization**. By instrumenting a program, we can determine, for any test case, which statements in that program were exercised (covered) by that test case. We can then prioritize test cases in terms of the total number of statements they cover by counting the number of statements covered by each test case and then sorting the test cases in descending order of that number. (When multiple test cases cover the same number of statements, an additional rule is necessary to order these test cases; we order them randomly.)

To illustrate, Fig. 1 depicts a procedure (left) and the statement coverage of the executable statements in that procedure achieved by three test cases (center). Applied in this case, total statement coverage prioritization yields test case order (3, 1, 2).

For a test suite containing m test cases and a program containing n statements, total statement coverage prioritization can be accomplished in time $O(m\,n + m\,log\,m)$. (The first term denotes the time required to count the statements covered by each test case, and the second term denotes the time required to sort the test cases according to coverage.) Typically, n is greater than m, in which case the cost of this prioritization is $O(m\,n)$.

Note that our measure of total statement coverage does not consider repetition in coverage in its calculation. That is, a statement that is executed once is treated the same as a statement that, due to looping, is executed multiple times. This treatment, however, is the treatment that underlies code-coverage-based testing techniques generally. Alternative measures could consider execution counts; we leave the investigation of such alternatives as a subject for future work.

M_5: **Additional statement coverage prioritization**. Total statement coverage prioritization schedules test cases in the order of total coverage achieved; however, having executed a test case and covered certain statements, more may be gained in subsequent testing by executing statements that have not yet been covered. Additional statement coverage prioritization iteratively selects a test case that yields the greatest statement coverage, then adjusts the coverage

```
   procedure P
1. s1
2. while (c1) do
3.   if (c2) then
4.     exit
     else
5.     s3
     endif
6.   s4
   endwhile
7. s5
8. s6
9. s7
```

STATEMENT COVERAGE

statement	test case 1	test case 2	test case 3
1	X	X	X
2	X	X	X
3		X	X
4		X	
5			X
6			X
7	X		X
8	X		X
9	X		X

BRANCH COVERAGE

	test case 1	test case 2	test case 3
entry	X	X	X
2-true		X	X
2-false	X		X
3-true		X	
3-false			X

Fig. 1. Procedure P and the statement and branch coverage of P achieved by three test cases.

information on all remaining test cases to indicate their coverage of statements not yet covered and repeats this process until all statements covered by at least one test case have been covered. (When multiple test cases cover the same number of statements not yet covered, an additional rule is necessary to choose one of these test cases; we do this randomly.)

Having ordered a subset of the test cases in a test suite in this manner, we may reach a point where each statement has been covered by at least one test case, and the remaining unprioritized test cases cannot add additional statement coverage. Next, we could order these remaining test cases using any prioritization technique; in this work, we order the remaining test cases by reapplying additional coverage prioritization (i.e., by resetting the coverage vectors for all of these test cases to their initial values, and reapplying the algorithm ignoring all previously prioritized test cases).

For illustration, consider Fig. 1. In this example, both total and additional statement coverage prioritization select test case 3 first; however, whereas total coverage prioritization selects test case 1 second, additional coverage prioritization detects that test case 1 covers no statements not already covered by test case 3 and that test case 2 covers an uncovered statement and outputs test case order (3, 2, 1).

Additional statement coverage prioritization requires coverage information for each unprioritized test case to be updated following the choice of each test case. Given a test suite containing m test cases and a program containing n statements, selecting a test case and readjusting coverage information has cost $O(m\,n)$ and this selection and readjustment must be performed $O(m)$ times. Therefore, the cost of additional statement coverage prioritization is $O(m^2\,n)$, a factor of m more expensive than total statement coverage prioritization.

M_6: **Total branch coverage prioritization**. Total branch coverage prioritization is the same as total statement coverage prioritization, except that it uses test coverage measured in terms of program branches rather than statements. In this context, we define *branch coverage* as coverage of each possible overall outcome of a (possibly compound) condition in a predicate. Thus, for example, each `if` or `while` statement must be exercised such that it evaluates at least once to `true` and at least once to `false`. To accommodate functions that contain no branches, we treat each function entry as a branch, and regard that branch as covered by each test case that causes the function to be invoked.

Because in theory branch coverage *properly subsumes* statement coverage [27] (e.g., a test suite that is adequate for branch coverage is necessarily adequate for statement coverage, but not vice-versa), one might conjecture that prioritization based on branch coverage should on average be at least as effective as, if not more effective than, prioritization based on statement coverage. On the other hand, the arms of a branch often contain different numbers of statements and, in this case, ordering by branches may cause less-than-ideal attention to be paid to branches that contain the most code; on this basis, one might conjecture that prioritization for statement coverage would be more effective than prioritization for branch coverage.[3] To begin to address these contradictory intuitions, empirical investigation of the relationship between statement- and branch-coverage-based prioritization techniques is necessary.

Fig. 1 (right) depicts the branch coverage achieved on the code depicted in the figure by the same three test cases used in the illustration of statement coverage prioritization. Applied to this example, total branch coverage prioritization outputs test case order (3, 2, 1).

M_7: **Additional branch coverage prioritization**. Additional branch coverage prioritization is the same as additional statement coverage prioritization, except that it uses test coverage measured in terms of program branches rather than statements. With this technique, too, we require a method for prioritizing the remaining test cases after complete coverage has been achieved and, in this work, we do this by resetting coverage vectors to their initial values and reapplying additional branch coverage prioritization to the remaining test cases.

3. This latter possibility was suggested by one of the anonymous reviewers.

Applied to the example and branch coverage information depicted in Fig. 1, total branch coverage prioritization outputs test case order (3, 2, 1). In this case, unlike the case with statement coverage prioritization, total and additional branch coverage prioritizations output identical test case orders.

M_8: **Total fault-exposing-potential (FEP) prioritization**. Statement and branch coverage prioritization consider only whether a statement or branch has been exercised by some test case. These techniques thus ignore a fact about test cases and faults: Some faults are more easily exposed than other faults, and some test cases are more adept at revealing particular faults than other test cases. More formally, the ability of a test case to expose a fault—that test case's *fault exposing potential* (FEP)—depends not only on whether the test case covers (executes) a faulty statement, but also on the probability that a fault in that statement will cause a failure for that test case [12], [14], [32], [33]. Although any practical determination of this probability must be an approximation, we wished to determine whether the use of such an approximation could yield a prioritization technique superior in terms of rate of fault detection than techniques based on simple code coverage.

Voas [33] provides one method for obtaining such approximations, in the form of *PIE* (propagation, infection, and execution) analysis. PIE analysis assesses the probability that, under a given input distribution, if a fault exists in a statement s, it will result in a failure. This probability, termed the *sensitivity* of s, is estimated by combining independent estimates of three probabilities: 1) the probability that s is executed (*execution probability*), 2) the probability that a change in s can cause a change in program state (*infection probability*), and 3) the probability that a change in state propagates to output (*propagation probability*). PIE analysis uses various methods to obtain these estimates: 1) simple code instrumentation to estimate execution probability, 2) a variant of *weak mutation* [18] in which syntactic changes are applied to s and then the state after s is examined for effects to estimate infection probability, and 3) state perturbation, in which the data state following s is altered and then program output is examined for differences to estimate propagation probability.

One approach to incorporating estimates of fault-exposing-potential would involve obtaining sensitivity estimates of the form suggested by Voas, and associating these estimates with test cases using test coverage information. For the purpose of test case prioritization, however, this approach has two disadvantages.

First, by factoring in execution probabilities, sensitivity measures the probability that a fault will cause a failure relative to an input distribution. When prioritizing test cases for regression testing based on existing coverage information, however, we are interested in the probability that, *if a test case executes* a statement s containing a fault, that fault will propagate to output. It is possible for s to have very high [low] infection and propagation probabilities with respect to the inputs that execute it, even though it has a very low [high] execution probability relative to an input distribution. Thus, the incorporation of execution probabilities into sensitivity estimates distorts the measure of the likelihood that a given test case that reaches s will expose a fault in s. For the application and approach that we consider, a more appropriate measure would consider only infection and propagation.

A second drawback of sensitivity in this context involves its treatment of propagation and infection estimates. Sensitivity analysis separately calculates these estimates and uses a conservative approach to combine them. This conservative approach is designed to reflect the worst case in which the set of data state errors that produce the infection estimate is exactly the set of data state errors that do not propagate to output, although, in general, this case may be unlikely to occur. This approach can result in low estimates of fault exposing potential, with a large number of statements receiving estimates of zero; these zero estimates may compromise the ability of test case prioritization techniques to create useful test case orderings.

Thus, in this work, to obtain an approximation of the fault-exposing-potential of a test case, we adopt an approach that uses mutation analysis [9], [13] to produce a combined estimate of propagation-and-infection that does not incorporate independent execution probabilities. (Mutation analysis creates a large number of faulty versions ("mutants") of a program by altering program statements, and uses these to assess the quality of test suites by measuring whether those test suites can detect those faults ("kill" those mutants).)

The approach works as follows: Given program P and test suite T, we first create a set of mutants $N = \{n_1, n_2, \ldots, n_m\}$ for P, noting which statement s_j in P contains each mutant. Next, for each test case $t_i \in T$, we execute each mutant version n_k of P on t_i, noting whether t_i kills that mutant. Having collected this information for every test case and mutant, we consider each test case t_i and each statement s_j in P, and calculate the fault-exposing-potential $FEP(s,t)$ of t_i on s_j as the ratio of mutants of s_j killed by t_i to the total number of mutants of s_j. Note that if t_i does not execute s_j, this ratio is zero.

To perform total FEP prioritization, given these $FEP(s,t)$ values, we next calculate, for each test case $t_i \in T$, an *award value*, by summing the $FEP(s_j, t_i)$ values for all statements s_j in P. Given these award values, we then prioritize test cases by sorting them in order of descending award value (resolving ties by random selection).

To illustrate, Fig. 2 depicts the procedure P considered in our earlier discussion of coverage-based prioritization techniques and a table listing fault-exposing-potential estimates that might be calculated for the three test cases and the statements in that procedure. In this case, the award value for test case t1 is 2.3, the award value for test case t2 is 2.41, the award value for test case t3 is 2.2, and total FEP prioritization outputs test case order (2, 1, 3).

Total FEP prioritization may appear, like statement- and branch-coverage-based prioritization, to ignore multiple statement executions caused by looping. However, because the mutation scores with which we obtain FEP values are obtained through actual test executions, they have captured at least some of the effects of looping on fault detection.

```
   procedure P
1. s1
2. while (c1) do
3.   if (c2) then
4.     exit
     else
5.     s3
     endif
6.   s4
   endwhile
7. s5
8. s6
9. s7
```

FEP(s,t) values

statement	test case 1	test case 2	test case 3
1	.4	.5	.3
2	.5	.9	.4
3		.01	.4
4		1.0	
5			.4
6			.1
7	.5		.2
8	.6		.3
9	.3		.1

Fig. 2. Procedure P and $FEP(s,t)$ values for three test cases.

One issue to consider with respect to the use of program mutation to approximate FEP values involves the "equivalent mutant" problem: The problem of determining whether a mutant version of a program is semantically equivalent to the original program. A semantically equivalent mutant can never be killed by any test case. The problem of identifying equivalent mutants is undecidable in general and, in practice, can involve considerable human effort. It was not feasible in our empirical studies to identify these mutants given the enormous numbers of mutants involved (over 160,000 mutants). Therefore, we considered two approaches for coping with the possible presence of these mutants.

The first approach is to consider mutants not killed by any test case used in the empirical studies to be semantically equivalent mutants, and ignore these mutants in our FEP calculations. (The number of test cases used in our empirical studies is also enormous, as we report in Section 3.4.) This approach, however, may overestimate the number of semantically equivalent mutants, and cause us to overestimate FEP values. Such overestimates may cause us to assign an inordinately high award value to any test case that executes statements containing such mutants—an award value that proclaims the test case more powerful than it is.

The second approach is to treat all mutants not killed by any test case as possibly nonequivalent and consider those mutants in our FEP calculations. This approach may underestimate the number of semantically equivalent mutants and cause us to underestimate FEP values. Such underestimates may cause us to assign an inordinately low award value to any test case that executes statements containing such mutants—an award value that proclaims the test case less powerful than it is.

We chose the second approach due to its conservatism.

Given the $FEP(s,t)$ values for a test suite containing m test cases and a program containing n statements, total FEP prioritization can be accomplished in time $O(m\,n + m\,log\,m)$. In general, n is greater than m, in which case, the cost of this prioritization is $O(m\,n)$, a worst-case time analogous to that for total statement coverage prioritization.

The cost of obtaining $FEP(s,t)$ values could, however, be quite high: Certainly, if these values are obtained through mutation analysis, this cost may be excessive. Thus, whereas our investigation of coverage-based prioritization techniques involves techniques that are potentially practical and applicable as presented, our investigation of FEP-based techniques should be considered exploratory. Such an exploration, however, is easily motivated: If FEP prioritization shows promise, this would justify a search for more cost-effective techniques for approximating fault-exposing potential, such as techniques that use constrained mutation [23], or techniques that use static measures of the likelihood of fault exposure [22].

M_9: **Additional fault-exposing-potential (FEP) prioritization**. Analogous to the extensions made to total statement and branch coverage prioritization to yield additional statement and branch coverage prioritization, we extend total FEP prioritization to create additional fault-exposing-potential (FEP) prioritization. This lets us account for the fact that additional executions of a statement may be less valuable than initial executions.

To describe this technique more precisely, we require a mechanism for measuring the value of an execution of a statement, that can be related to FEP values. For this, we use the term *confidence*. We say that the confidence in statement s, $C(s)$, is an estimate of the probability that s is correct. ($C(s)$ is a value between 0 and 1, inclusive.) If we execute a test case t that exercises s and does not reveal a fault in s, $C(s)$ should increase. Assume that, prior to execution of t, the confidence in statement s is $C(s)$, and the fault-exposing potential of t for s is $FEP(s,t)$. Then, after execution of t and if t exposes no fault in s, our new confidence in s, $C'(s)$, is

$$C'(s) = 1 - (1 - C(s)) \cdot (1 - FEP(s,t)).$$

Simplifying this equation, we obtain

$$C'(s) = C(s) + (1 - C(s)) \cdot FEP(s,t).$$

So, the additional confidence in statement s that we gain by executing test case t through s is

$$C_{addi}(s) = C'(s) - C(s) = (1 - C(s)) \cdot FEP(s,t).$$

Calculation 1:

C_addi(s) values per test case

statement	test case 1	test case 2	test case 3
1	.4	.5	.3
2	.5	.9	.4
3		.01	.4
4		1.0	
5			.4
6			.1
7	.5		.2
8	.6		.3
9	.3		.1

Calculation 2:

C(s) after execution of test case 2

statement	
1	.5
2	.9
3	.01
4	1.0
5	0
6	0
7	0
8	0
9	0

Calculation 3:

C_addi(s) values per test case

statement	test case 1	test case 2	test case 3
1	.2		.15
2	.05		.04
3			.40
4			
5			.4
6			.1
7	.5		.2
8	.6		.3
9	.3		.1

Fig. 3. Values calculated during additional FEP prioritization for the program and test cases of Fig. 2.

We define $C_{addi}(t)$, the additional confidence gained from executing test case t on program P, as the sum of the $C_{addi}(s)$ for all statements s covered by t. Thus, if $s_1, s_2, \cdots, s_k$ are statements covered by t, then

$$C_{addi}(t) = C_{addi}(s_1) + C_{addi}(s_2) + \cdots + C_{addi}(s_k).$$

Additional FEP prioritization iteratively selects a test case t that yields the greatest $C_{addi}(t)$ value given the current $C_{addi}(s)$ values, then updates the $C(s)$ values for statements covered by t and recalculates the $C_{addi}(s)$ values of remaining statements for remaining test cases based on the updated $C(s)$ values and then repeats this process until all test cases have been prioritized.

Although, in practice, the initial values of $C(s)$ could be set differently for different statements, we initialize all $C(s)$ to a fixed value. The fixed value we chose is 0, which implies that we have no confidence in any statement prior to running the test suite. (We could choose other initial values. For example, 0.5 could be used to indicate that the probabilities of a statement being correct and containing a fault are equal.)

As an example of the application of additional FEP prioritization, again consider Fig. 2. We initialize $C(s)$ to 0 for all statements. In this case, the $C_{addi}(s)$ values that would result from executing each test case are shown in Fig. 3 (Calculation 1) and are (for each test case) equivalent to the original $FEP(s,t)$ values. Thus, we have $C_{addi}(t_1) = 2.3$, $C_{addi}(t_2) = 2.41$, and $C_{addi}(t_3) = 2.2$, and additional FEP prioritization, like total FEP prioritization, selects test case 2 as the first test case.

Having chosen this test case, additional FEP prioritization now calculates, for each statement s, $C'(s)$, the new confidence in that statement. Because test case 2 executes only statements 1, 2, 3, and 4, their confidence values increase while the confidence values of other statements remain 0. Fig. 3 (Calculation 2) shows the resulting values. Next, $C_{addi}(s)$ values are recalculated for each remaining test case as shown in the figure (Calculation 3); only the values for statements 1, 2, and 3 are altered. From these, we calculate $C_{addi}(t_1) = 1.65$ and $C_{addi}(t_3) = 1.69$. Additional FEP prioritization selects test case 3 next because it yields the greatest gain in confidence. The technique outputs test case order (2, 3, 1), in which the order of the second and third test cases is the reverse of the order output by total FEP prioritization.

One difference between additional FEP prioritization and additional statement or branch coverage prioritization is that, in the additional FEP prioritization algorithm, we are not likely to need to check whether "full confidence" has been achieved: It is not likely that we will reach a point at which no additional confidence can be gained for all remaining test cases. The reason for this is that, for a test case t's $C_{addi}(t)$ to be 0, the $C(s)$ for each statement covered by t must be 1, and for the $C(s)$ for a statement to be 1, there must exist some test case t' for which $FEP(s,t')$ is 1. $FEP(s,t)$ may be estimated to be 1 in some cases, but it is unlikely that it will be estimated to be 1 for each statement covered by t. If this unlikely event did occur, we could proceed as with other "additional" coverage prioritization techniques, resetting $C(s)$ and $C_{addi}(t)$ values to their initial states for those test cases not yet prioritized and reapplying the algorithm to those test cases; however, in our empirical studies, this event did not occur.

Like additional statement coverage prioritization, additional FEP prioritization requires coverage information for each unprioritized test case to be updated following the choice of each test case. Therefore, its cost, for a test suite of m test cases and a program containing n statements, is $O(m^2\, n)$, a factor of m more expensive than total FEP prioritization. Also, like total FEP prioritization, however, additional FEP prioritization requires a method for estimating FEP values, a potentially expensive requirement.

2.2 Related Work

In [1], Avritzer and Weyuker present techniques for generating test cases that apply to software that can be modeled by Markov chains, provided that operational profile data is 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 probability mass earlier in testing, essentially, prioritizing the test cases in an order that increases the likelihood that

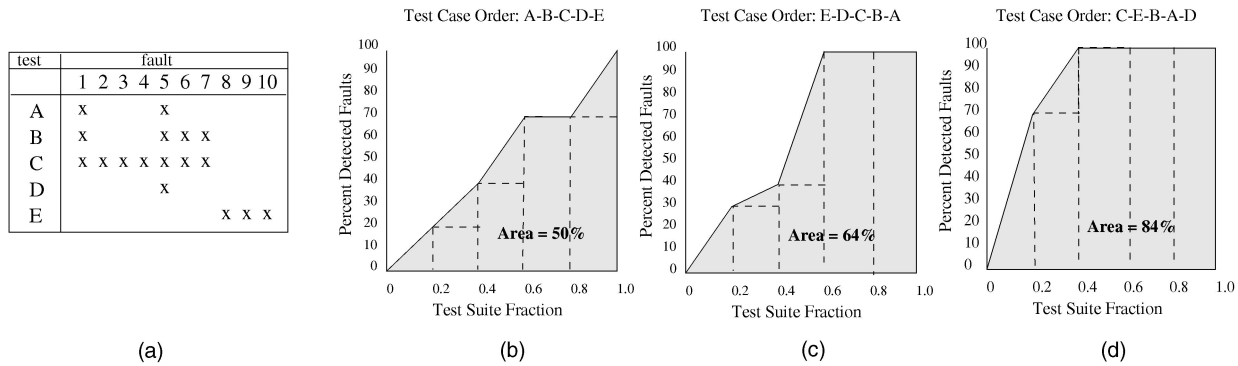


Fig. 4. Example illustrating the APFD measure. (a) Test suite and faults exposed. (b) APFD for Prioitized Suite T1. (c) APFD for Prioritized Suite T2. (d) APFD for Prioritized Suite T3.

faults more likely to be encountered in the field will be uncovered earlier in testing. The approach provides an example of the application of prioritization to the initial testing of software, when test suites are not yet available.

In [36], Wong et al. suggest prioritizing test cases according to the criterion of "increasing cost per additional coverage." Although not explicitly stated by the authors, one possible goal of this prioritization is to reveal faults earlier in the testing process. The authors restrict their attention, however, to prioritization of test cases for execution on a specific modified version of a program (what we have termed "version-specific prioritization") and to prioritization of only the subset of test cases selected by a safe regression test selection technique from the test suite for the program. The authors do not specify a mechanism for prioritizing the remaining test cases after full coverage has been achieved. The authors describe a case study in which they applied their technique to a program of over 6,000 lines of executable code (the same program, `space`, that we use in two of the empirical studies reported in this paper), and evaluated the resulting test suites against 10 faulty versions of that program. They conclude that the technique was cost-effective in that application.

3 Empirical Studies of Test Case Prioritization Techniques

To investigate test case prioritization and to compare and evaluate the test case prioritization techniques described in Section 2, we performed several empirical studies.[4] This section describes those studies, including design, measures, subjects, results, and threats to validity.

3.1 Research Questions

We are interested in the following research questions.

- [**Q1:**] Can test case prioritization improve the rate of fault detection of test suites?
- [**Q2:**] How do the various test case prioritization techniques presented in Section 2 compare to one another in terms of effects on rate of fault detection?

4. The subjects (programs, program versions, test cases, and test suites) used in these studies and the data sets collected can be obtained by contacting the first author.

3.2 Effectiveness Measure

To address our research questions, we require a measure with which to assess and compare the effectiveness of various test case prioritization techniques. (In terms of Definition 1, this measure plays the role of the function f.) As a measure of how rapidly a prioritized test suite detects faults, we use a weighted average of the percentage of faults detected, or *APFD*, during the execution of the test suite. These values range from 0 to 100; higher APFD numbers mean faster (better) fault detection rates.

To illustrate this measure, consider an example program with 10 faults and a test suite of five test cases, **A** through **E**, with fault detecting abilities, as shown in Fig. 4a.

Suppose we place the test cases in order **A–B–C–D–E** to form a prioritized test suite $T1$. Fig. 4b shows the percentage of detected faults versus the fraction of the test suite $T1$ used. After running test case **A**, two of the 10 faults are detected; thus, 20 percent of the faults have been detected after 0.2 of test suite $T1$ has been used. After running test case **B**, two more faults are detected and, thus, 40 perent of the faults have been detected after 0.4 of the test suite has been used. In Fig. 4b, the area inside the inscribed rectangles (dashed boxes) represents the weighted percentage of faults detected over the corresponding fraction of the test suite. The solid lines connecting the corners of the inscribed rectangles interpolate the gain in the percentage of detected faults.[5] The area under the curve thus 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 measure (APFD); the APFD is 50 percent in this example.

Fig. 4c reflects what happens when the order of test cases is changed to **E–D–C–B–A**, yielding test suite $T2$, a "faster detecting" suite than $T1$ with an APFD of 64 percent. Fig. 4d shows the effects of using a prioritized test suite $T3$ whose test case ordering is **C–E–B–A–D**. By inspection, it is clear that this ordering results in the earliest detection of the most faults and illustrates an optimal ordering with an APFD of 84 percent.

5. This interpolation is a granularity adjustment when only a small number of test cases comprise a test suite (the larger the test suite the smaller this adjustment). The interpolation also corresponds to an interpretation under which, as each test case in the test suite executes, progress is considered to be made toward detecting the faults that are detected by that test case.

TABLE 3
Experiment Subjects

Program	Lines of Executable Code	Number of Versions	Number of Mutants	Test Pool Size	Average Test Suite Size
print_tokens	402	7	4030	4130	16
print_tokens2	483	10	4346	4115	12
replace	516	32	9622	5542	19
schedule	299	9	2153	2650	8
schedule2	297	10	2822	2710	8
tcas	138	41	2876	1608	6
tot_info	346	23	5898	1052	7
space	6218	35	132163	13585	155

Note that our measure of effectiveness, APFD, does not incorporate factors of the cost of performing prioritization, and we do not measure such factors in our experiments. There are reasons for this. Our implementations of techniques were not built for efficiency, and our studies required us to run processes continuously on several machines over several weeks, during which time we were unable to control for other processes using the hardware and, thereby, altering timings. It is not clear that performance-cost measurements obtained from such tools run under such conditions would be meaningful.

Moreover, any cost-benefits trade-offs involving test case prioritization depend upon the testing process in use, and how test case prioritization fits into that process. With general test case prioritization (the variety of prioritization that we are investigating), prioritization can be performed "offline," following a release of a system, at a time when resource usage may be noncritical (provided it falls below a certain threshold). The cost of performing this prioritization can then be amortized over successive releases of the software. The cost-benefit trade-offs of using such prioritization techniques will vary with the process used and the resources available, and a single measure incorporating both costs and benefits could obscure cost-effectiveness analyses that might apply under particular processes.

Thus, instead of measuring and reporting run-time costs, we have provided overall complexity analyses of test case prioritization techniques, and we use these in Section 4 when we discuss practical implications of our results.

3.3 Prioritization and Analysis Tools

To perform our empirical studies, we required several tools. To obtain test coverage and control-flow graph information, we used the Aristotle program analysis system [16]. To obtain mutation scores for use with FEP prioritization, we used the Proteum mutation system [8]. We created prioritization tools that implement the techniques outlined in Section 2.

3.4 Subjects

We used eight C programs as subjects (see Table 3). The first seven programs with faulty versions and test cases were assembled by researchers at Siemens Corporate Research for a study of the fault detection capabilities of control-flow and data-flow coverage criteria [19]. We refer to these as the *Siemens programs*. The eighth program is a program developed for the European Space Agency. We refer to this program as `space`. We further discuss the Siemens programs and `space` in the following sections.

3.4.1 Siemens Programs, Versions, and Test Suites

The Siemens programs perform a variety of tasks: `tcas` is an aircraft collision avoidance system, `schedule2` and `schedule` are priority schedulers, `tot_info` computes statistics given input data `print_tokens` and `print_tokens2` are lexical analyzers, and `replace` performs pattern matching and substitution.

The researchers at Siemens sought to study the fault detecting effectiveness of coverage criteria. Therefore, they created faulty versions of the seven base programs by manually seeding those programs with faults, usually by modifying a single line of code in the program. Their goal was to introduce faults that were as realistic as possible, based on their experience with real programs. Ten people performed the fault seeding, working "mostly without knowledge of each other's work" [19, p. 196]. The result of this effort was between seven and 41 versions of each base program (see Table 3), each containing a single fault.

In this context, the use of single-fault versions is an important experiment design choice that allows experimenters to precisely determine whether a test case reveals a particular fault simply by determining whether the version containing that fault fails. In the absence of this methodology, it may be difficult or impossible to associate test cases with particular faults. This choice does, however, pose a potential threat to validity; we discuss this further in Section 3.6.

For each base program, the researchers at Siemens created a large test pool containing possible test cases for the program. To populate these test pools, they first created an initial suite of black-box test cases "according to good testing practices, based on the tester's understanding of the program's functionality and knowledge of special values and boundary points that are easily observable in the code" [19, p. 194], using the *category partition method* and the Siemens Test Specification Language tool [2], [25]. They then augmented this suite with manually-created white-box test cases to ensure that each executable statement, edge, and definition-use pair in the base program or its control-flow graph was exercised by at least 30 test cases. To obtain

meaningful results with the seeded versions of the programs, the researchers retained only faults that were "neither too easy nor too hard to detect" [19, p. 196], which they defined as being detectable by at most 350 and at least three test cases in the test pool associated with each program.

To obtain sample test suites for these programs, we used the test pools for the base programs and test-coverage information about the test cases in those pools to generate 1,000 branch-coverage-adequate test suites for each program. More precisely, to generate a test suite T for base program P from test pool T_p, we used the C pseudo-random-number generator `rand`, seeded initially with the output of the C `time` system call, to obtain integers that we treated as indexes into T_p (modulo $|T_p|$). We used these indexes to select test cases from T_p; we added a selected test case t to T only if t added to the cumulative branch coverage of P achieved by the test cases added to T thus far. We continued to add test cases to T until T contained at least one test case that would exercise each executable branch in the base program. Table 3 lists the average sizes of the branch-coverage-adequate test suites generated by this procedure for the subject programs.

Using Proteum, we generated mutants for the Siemens programs. Table 3 reports the numbers of mutant programs thus created.

3.4.2 Space, Versions, and Test Suites

`Space` consists of 9,564 lines of C code (6,218 executable), and functions as an interpreter for an array definition language (ADL). The program reads a file that contains several ADL statements, and checks the contents of the file for adherence to the ADL grammar and to 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.

`Space` has 33 associated versions, each containing a single fault that had been discovered during the program's development. (We adopt the single-fault-version approach used with the Siemens programs for `space`, for the same reasons.) Through working with this program, we discovered five additional faults, and created versions containing just those faults. We also discovered that three of the "faulty versions" originally supplied were actually semantically equivalent to the base version. We excluded these from our study; therefore, we ultimately used 35 faulty versions.

We constructed a test pool for `space` in two stages. We obtained an initial pool of 10,000 test cases from Vokolos and Frankl; they had created this pool for another study by randomly generating test cases [35]. Beginning with this initial pool, we instrumented the program for coverage and then added additional test cases to the pool until it contained, for each executable statement or edge (though unlike the Siemens programs, not for each definition-use pair) in the program or its control flow graph, at least 30 test cases that exercised that statement or edge.[6] This process yielded a test pool of 13,585 test cases.

6. We treated the statements and edges executable only on failure of one of the seventeen `malloc` calls found in the program as nonexecutable.

We used `space`'s test pools to obtain branch-coverage-adequate test suites for the program, following the same process used for the Siemens programs. The resultant test suites ranged in size from 141 to 169 test cases, averaging 155 test cases. Initially, we generated 1,000 such test suites. Due to the time required to exercise the mutants of `space` on all of the test cases contained in these 1,000 test suites, we randomly sampled these 1,000 test suites, selecting 50 test suites to use in our studies. This selection allowed us to restrict our mutation analysis to the 4,898 test cases contained in the selected suites.

As with the Siemens programs, we used Proteum to generate mutants for `space`; the tool produced 132,163 mutants.

3.5 Empirical Studies and Results

We performed four empirical studies, in which we varied the subject programs and the faults used. We next discuss each of these studies in turn, presenting their results and initial analysis. We provide further discussion of the results and their practical implications in Section 4.

3.5.1 Study 1: Siemens Programs with APFD Measured Relative to Siemens Faults

In our first study, we investigated the application of prioritization techniques to the Siemens programs, measuring APFD relative to the set of faults provided with those programs.

For each subject program P, we applied prioritization techniques M_2 through M_9 to each of the 1,000 sample test suites, thus obtaining 8,000 prioritized test suites. We retained the original 1,000 test suites (untreated) as controls; for analysis, we considered these "prioritized" by technique M_1. We calculated the APFD values of these 9,000 prioritized test suites, relative to the faults provided with the programs, and used these as the statistical data sets.

An initial indication of how each prioritization technique affected a test suite's rate of detection in this study can be determined from Fig. 5, which presents boxplots[7] of the APFD values of the nine categories of prioritized test suites for each program and an all-program total. M_1 is the control group. M_2 is the random prioritization group. M_3 is the optimal prioritization group. Comparing the boxplots of M_3 to those of M_1 and M_2, it is readily apparent that optimal prioritization greatly improved the rate of fault detection (i.e., increased APFD values) of the test suites in comparison to no prioritization and random prioritization. Examining the boxplots of the other prioritization techniques, M_4 through M_9, it seems that all produce some improvement. However the overlap in APFD values mandates formal statistical analysis.

Using the SAS statistical package [10] to perform an ANOVA analysis,[8] we were able to reject the null hypothesis that the APFD means for the various techniques were equal ($\alpha = .05$), confirming our boxplot observations. However, the ANOVA analysis indicated statistically

7. Boxplots provide a concise display of a distribution. The central line in each box marks the median value. The edges of the box mark the first and third quartiles. The whiskers extend from the quartiles to the farthest observation lying within 1.5 times the distance between the quartiles. Individual markers beyond the whiskers are outliers.

8. ANOVA is an acronym for ANalysis Of VAriance, a standard statistical technique that is used to study the variability of experimental data [17].

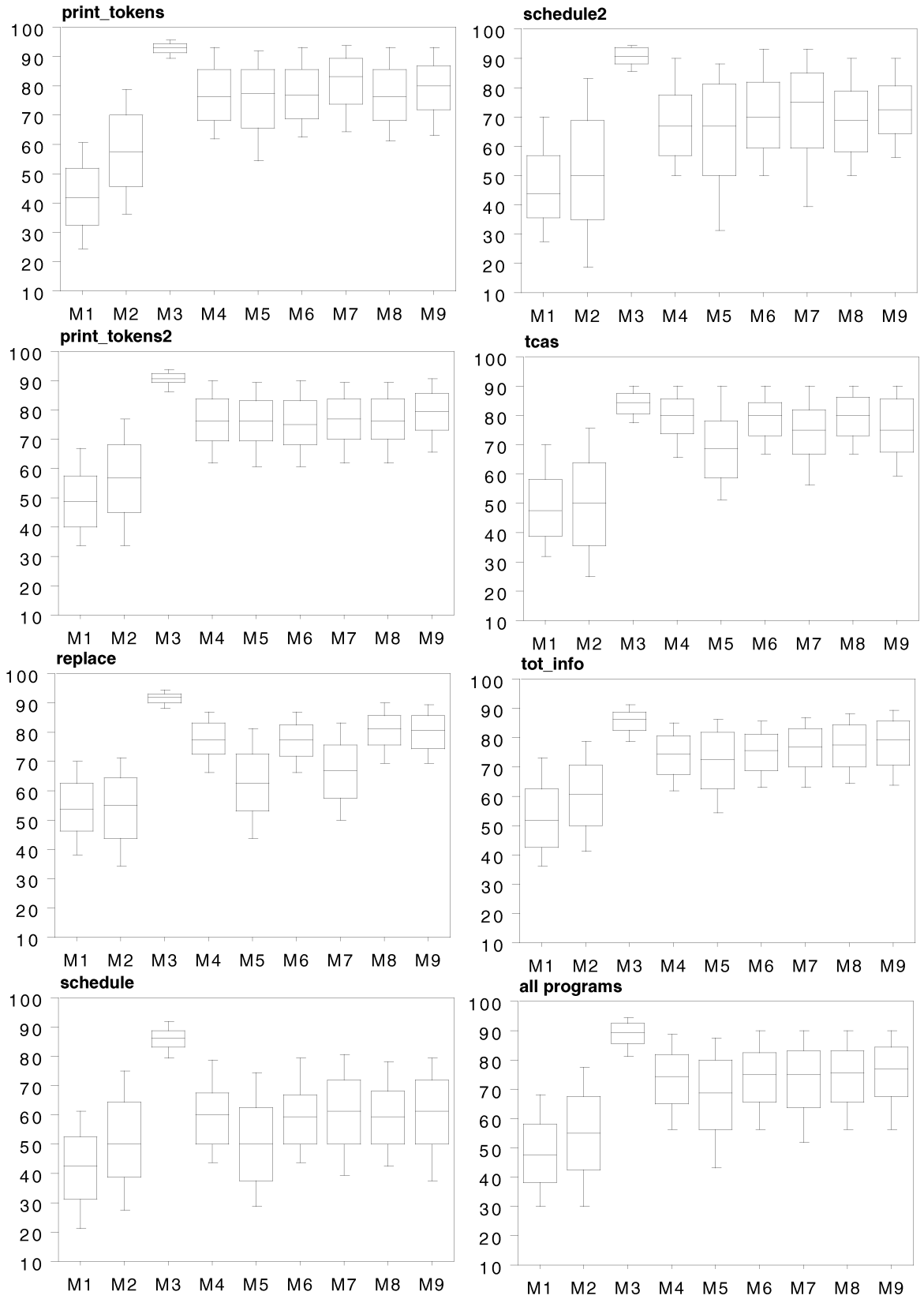


Fig. 5. APFD boxplots for Study 1 (vertical axis is APFD score): By program and by technique. The techniques are: M_1: untreated, M_2: random, M_3: optimal, M_4: stmt-total, M_5: stmt-addtl, M_6: branch-total, M_7: branch-addtl, M_8: FEP-total, and M_9: FEP-addtl.

significant cross-factor interactions: programs *have* an effect on APFD values. Thus, general statements about technique effects must be qualified.

While rejection of the null hypothesis tells us that some techniques produce statistically different APFD means, to determine which techniques differ from each other requires

TABLE 4
Bonferroni Means Separation Tests for Study 1

print_tokens			
Grouping		Mean	Technique
	A	92.5461	optimal
	B	80.8842	brch-addtl
	C	78.2727	FEP-addtl
D	C	76.8573	brch-total
D	E	76.4770	FEP-total
D	E	76.4647	stmt-total
	E	74.8199	stmt-addtl
	F	57.2829	random
	G	42.6163	untreated
df= 8991 MSE= 155.0369 Critical Value of T= 3.20			
Minimum Significant Difference= 1.7808 (α=.05)			

schedule2			
Grouping		Mean	Technique
	A	90.1794	optimal
	B	72.0518	FEP-addtl
	B	70.6432	brch-total
C	B	70.2513	brch-addtl
C	D	68.0438	FEP-total
	D	67.5409	stmt-total
	E	63.7391	stmt-addtl
	F	51.3077	random
	G	47.0302	untreated
df= 8127 MSE= 280.635 Critical Value of T= 3.20			
Minimum Significant Difference= 2.5199 (α=.05)			

print_tokens2			
Grouping		Mean	Technique
	A	90.5152	optimal
	B	78.3211	FEP-addtl
	C	76.1678	brch-addtl
	C	75.8848	stmt-total
	C	75.7985	FEP-total
	C	75.5995	stmt-addtl
	C	74.8830	brch-total
	D	55.9729	random
	E	49.3272	untreated
df= 8991 MSE= 124.203 Critical Value of T= 3.20			
Minimum Significant Difference= 1.5939 (α=.05)			

tcas			
Grouping		Mean	Technique
	A	83.8845	optimal
	B	78.9253	stmt-total
	B	78.7998	FEP-total
	B	78.5781	brch-total
	C	75.1880	FEP-addtl
	D	73.3552	brch-addtl
	E	68.5357	stmt-addtl
	F	50.1038	random
	F	49.4311	untreated
df= 8973 MSE= 148.5302 Critical Value of T= 3.20			
Minimum Significant Difference= 1.7447 (α=.05)			

replace			
Grouping		Mean	Technique
	A	91.6901	optimal
	B	80.0171	FEP-total
	B	79.6959	FEP-addtl
	C	77.1355	stmt-total
	C	76.8482	brch-total
	D	66.5639	brch-addtl
	E	62.3795	stmt-addtl
	F	54.4460	untreated
	F	54.0668	random
df= 8991 MSE= 110.782 Critical Value of T= 3.20			
Minimum Significant Difference= 1.5053 (α=.05)			

tot_info			
Grouping		Mean	Technique
	A	85.4258	optimal
	B	77.5442	FEP-addtl
C	B	76.8218	FEP-total
C	D	75.8798	brch-addtl
E	D	74.8807	brch-total
E		73.9979	stmt-total
	F	71.4503	stmt-addtl
	G	60.0587	random
	H	53.1124	untreated
df= 8991 MSE= 110.4918 Critical Value of T= 3.20			
Minimum Significant Difference= 1.5033 (α=.05)			

schedule			
Grouping		Mean	Technique
	A	85.7074	optimal
	B	60.6765	brch-addtl
	B	59.8694	stmt-total
	B	59.8484	FEP-addtl
	B	59.6161	brch-total
	B	59.4430	FEP-total
	C	51.4087	random
	C	50.4418	stmt-addtl
	D	41.9670	untreated
df= 8991 MSE= 222.3662 Critical Value of T= 3.20			
Minimum Significant Difference= 2.1327 (α=.05)			

All Programs			
Grouping		Mean	Technique
	A	88.5430	optimal
	B	74.4501	FEP-addtl
	C	73.7049	FEP-total
D	C	73.2205	brch-total
D		72.9030	stmt-total
	E	71.9919	brch-addtl
	F	66.7502	stmt-addtl
	G	54.3575	random
	H	48.2927	untreated
df= 62055 MSE= 162.9666 Critical Value of T= 3.20			
Minimum Significant Difference= 0.6948 (α=.05)			

running a multiple-comparison procedure [26]. Of the commonly used means separation tests, we elected to use the Bonferroni method [17]—for its conservatism and generality.

Using Bonferroni, we calculated the minimum statistically significant difference between APFD means for each program. These are given in Table 4. The techniques are listed within each program subtable by their APFD mean values, from higher (better) to lower (worse). Grouping letters partition the techniques; techniques that are not significantly different share the same grouping letter.

Examination of these subtables affirms what the boxplots indicate: All of the noncontrol techniques provided some significant improvement in rate of fault detection in comparison to no prioritization and random prioritization.

Although the relative improvement provided by each technique is dependent on the program, the *All Programs* subtable does show that additional FEP prioritization performed better overall than other techniques, and that total FEP prioritization performed better than all but branch-total prioritization (and no worse than branch-total). Also, the *All Programs* subtable suggests that branch-coverage-based techniques performed as well as or better

than their corresponding statement-coverage-based techniques (e.g., branch-total performed as well as statement-total, and branch-additional outperformed statement-additional).

It is also interesting that, in all but one case (print_tokens), total branch coverage prioritization performed as well as or outperformed additional branch coverage prioritization and, in all cases, total statement coverage prioritization performed as well as or outperformed additional statement coverage prioritization. Another effect worth noting is that on five of the seven programs and, overall, randomly prioritized test suites outperformed untreated test suites. We comment further on these effects in Section 4.

3.5.2 Study 2: Siemens Programs with APFD Measured Relative to Mutants

One of the threats to external validity of our first empirical study is that the faulty versions provided with the Siemens programs represent only a small subset of the faults that might occur in practice in those programs. (We further discuss threats to the validity of our studies in Section 3.6.) This threat can be addressed only by performing additional studies using additional varieties of faults. As a first step in this direction, in our second study, we investigated the application of prioritization techniques to the Siemens programs, measuring APFD *relative to the set of mutant versions of those programs*.

The study used the same design as Study 1. For each subject program P, we applied the prioritization techniques M_2 through M_9 to each of the 1,000 sample test suites, obtaining 8,000 prioritized test suites. Again, we retained the 1,000 original test suites (untreated) as controls; for analysis, we considered these "prioritized" by technique M_1. We then calculated the APFD values of these 9,000 prioritized test suites relative to the mutant versions of those programs and used these as the statistical data sets.[9] Note that each mutant version consisted of the base version with a single mutation applied. Thus, the column entitled "Number of Mutants" in Table 3 indicates the number of mutant versions considered: this number ranged from 2,153 on schedule to 9,622 on replace.

Fig. 6 presents boxplots of the APFD values of the nine categories of prioritized test suites for each program and an all-program total. The figure is similar to Fig. 5, but its APFD values are calculated based on different faulty versions for each base program (i.e., the mutant versions of the program). Table 5 presents the results of applying Bonferroni means separation tests to the data.

Examining Fig. 6 and Table 5, it is again apparent that all of the noncontrol techniques produce improvements in APFD values of test suites in comparison to no prioritization and random prioritization. Also, similar to Study 1, considering overall results, additional and total FEP prioritization outperformed all prioritization techniques other than optimal, but these results did vary somewhat across individual programs. Further, similar to Study 1, branch-coverage-based techniques almost always performed as well as or better than their corresponding statement-coverage-based techniques (the one exception being on tcas, where total statement prioritization outperformed total branch prioritization).

Again, as in Study 1, total statement coverage prioritization performed as well as or better than additional statement coverage overall. However, this relationship did not hold for branch coverage prioritization, where additional-branch coverage outperformed total-branch coverage overall. Finally, in this study, unlike Study 1, randomly prioritized test suites did not outperform untreated test suites.

3.5.3 Study 3: Space with APFD Measured Relative to Actual Faults

In our third empirical study, we investigated the application of prioritization techniques to space, measuring APFD relative to the set of actual faults provided with that program. We applied each of the prioritization techniques M_2 through M_9 to each of the 50 sample test suites, yielding 400 prioritized test suites. We again retained the original (untreated) 50 test suites as controls; for analysis, we considered these "prioritized" by technique M_1. We calculated the APFD values of these 450 prioritized test suites—relative to the actual faults provided with space—and used these as the statistical data sets.

Fig. 7 presents the boxplots of the APFD values for the nine categories of prioritized test suites for space. As in the earlier studies, we analyzed the differences between APFD means for the program: the results are given in Table 6.

Examining Fig. 7 and Table 6, it is again apparent that prioritization techniques M_3 through M_9 produced improvements in APFD values of test suites compared to random and no prioritization. Among the techniques, additional FEP prioritization outperformed the other techniques. There was no significant difference, however, among the four coverage-based techniques and total FEP prioritization. Also, in this study, as in Study 2, randomly prioritized test suites and untreated test suites were indistinguishable. However, unlike earlier studies, no distinctions can be made between statement-coverage-based and branch-coverage-based techniques, or among them, between total and additional-coverage-based variants.

3.5.4 Study 4: Space with APFD Measured Relative to Mutants

In our final study, we investigated the application of prioritization techniques to space, measuring APFD relative to the set of mutants of that program. Again, we applied the prioritization techniques M_2 through M_9 to each of the 50 sample test suites, obtaining 400 prioritized test suites. Again, we retained the original (untreated) 50 test suites as controls; for analysis, we considered these "prioritized" by technique M_1. We then calculated the APFD values of these 450 prioritized test suites *relative to the mutant versions of* space (132,163 versions—one for each mutant) and used these as the statistical data sets.

Examining Fig. 8 and Table 7, it is again apparent that prioritization techniques M_3 through M_9 produced improvements in APFD values of test suites. Again, additional

9. A reader familiar with mutation analysis may wonder whether the presence of equivalent mutants among the mutant versions of the Siemens programs would affect our APFD calculations. In fact, equivalent mutants have no effect on APFD calculations, because APFD calculations measure only the rate at which *detectable* faults are revealed by a test suite.

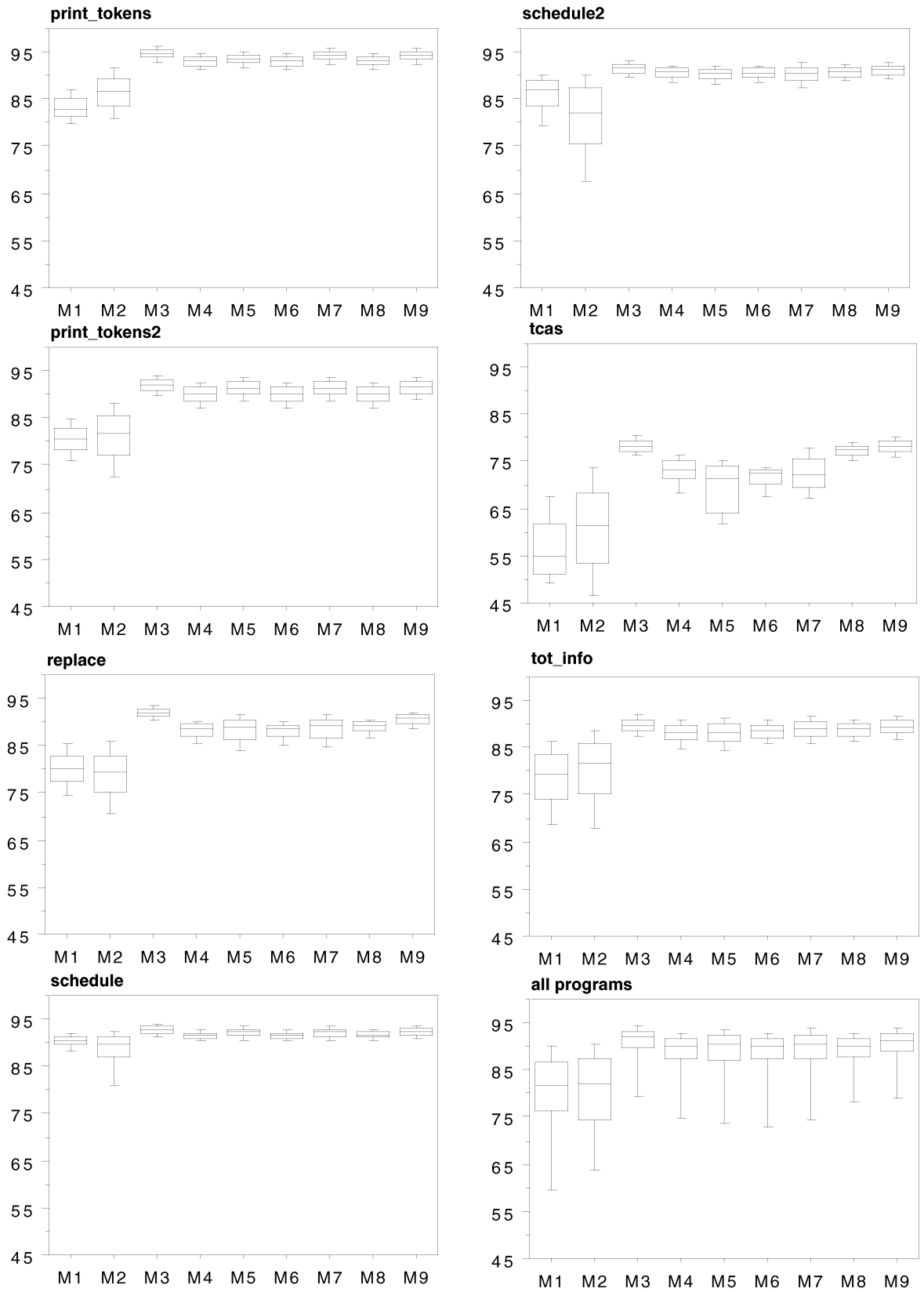


Fig. 6. APFD boxplots for Study 2 (vertical axis is APFD score): By program and by technique. The techniques are: M_1: untreated, M_2: random, M_3: optimal, M_4: stmt-total, M_5: stmt-addtl, M_6: branch-total, M_7: branch-addtl, M_8: FEP-total, and M_9: FEP-addtl.

FEP prioritization outperformed the other techniques. Also, as in Study 3, branch and statement-coverage-based techniques are indistinguishable; however, unlike in our first two studies, here additional-coverage-based techniques always outperform total-coverage-based techniques and by a relatively large margin.

TABLE 5
Bonferroni Means Separation Tests for Study 2

print_tokens			
Grouping		Mean	Technique
	A	94.5726	optimal
	B	94.1552	brch-addtl
	B	94.0983	FEP-addtl
	C	93.4522	stmt-addtl
	D	92.9773	brch-total
	D	92.9769	FEP-total
	D	92.9676	stmt-total
	E	86.3318	random
	F	83.0611	untreated
df= 8991 MSE= 4.08951 Critical Value of T= 3.20			
Minimum Significant Difference= 0.2892 (α=.05)			

print_tokens2			
Grouping		Mean	Technique
	A	91.8595	optimal
	B	91.3132	FEP-addtl
	B	91.2169	brch-addtl
	B	91.1878	stmt-addtl
	C	89.9208	stmt-total
	C	89.9189	FEP-total
	C	89.8104	brch-total
	D	80.9336	random
	E	80.3819	untreated
df= 8991 MSE= 8.62951 Critical Value of T= 3.20			
Minimum Significant Difference= 0.4201 (α=.05)			

replace			
Grouping		Mean	Technique
	A	92.0247	optimal
	B	90.4726	FEP-addtl
	C	88.8712	FEP-total
D	C	88.5446	brch-addtl
D	E	88.1150	stmt-addtl
	E	88.1000	stmt-total
	E	88.0152	brch-total
	F	80.0455	untreated
	F	78.6597	random
df= 8991 MSE= 9.03445 Critical Value of T= 3.20			
Minimum Significant Difference= 0.4299 (α=.05)			

schedule			
Grouping		Mean	Technique
	A	92.6215	optimal
B	A	92.3722	FEP-addtl
B	C	92.1101	stmt-addtl
	C	92.0307	brch-addtl
	D	91.6136	FEP-total
	D	91.5112	stmt-total
	D	91.4804	brch-total
	E	90.0178	untreated
	F	88.2226	random
df= 8991 MSE= 3.91423 Critical Value of T= 3.20			
Minimum Significant Difference= 0.283 (α=.05)			

schedule2			
Grouping		Mean	Technique
	A	91.4701	optimal
B	A	90.9788	FEP-addtl
B	C	90.7122	FEP-total
C	D	90.5303	stmt-total
C	D	90.3959	brch-total
C	D	90.2108	stmt-addtl
	D	90.0740	brch-addtl
	E	85.7386	untreated
	F	80.3034	random
df= 8991 MSE= 12.9149 Critical Value of T= 3.20			
Minimum Significant Difference= 0.514 (α=.05)			

tcas			
Grouping		Mean	Technique
	A	78.3102	optimal
	A	78.1003	FEP-addtl
	B	77.2512	FEP-total
	C	72.9459	stmt-total
	C	72.2930	brch-addtl
	D	71.5644	brch-total
	E	69.2863	stmt-addtl
	F	60.6226	random
	G	56.7793	untreated
df= 8991 MSE= 24.0915 Critical Value of T= 3.20			
Minimum Significant Difference= 0.702 (α=.05)			

tot_info			
Grouping		Mean	Technique
	A	89.6274	optimal
B	A	89.2970	FEP-addtl
B	C	88.7839	brch-addtl
	C	88.6767	FEP-total
D	C	88.3099	brch-total
D		87.9411	stmt-addtl
D		87.9338	stmt-total
	E	79.8922	random
	F	78.1495	untreated
df= 8991 MSE= 15.7992 Critical Value of T= 3.20			
Minimum Significant Difference= 0.56851 (α=.05)			

All Programs			
Grouping		Mean	Technique
	A	90.0694	optimal
	B	89.5189	FEP-addtl
	C	88.5744	FEP-total
	D	88.1569	brch-addtl
	E	87.7013	brch-total
	F	87.5076	stmt-total
	F	87.4719	stmt-addtl
	G	79.2808	random
	G	79.1676	untreated
df= 62055 MSE= 11.2104 Critical Value of T= 3.20			
Minimum Significant Difference= 0.1809 (α=.05)			

3.6 Threats to Validity

In this section, we discuss some of the potential threats to the validity of our studies. There are three types of threats to consider:

- threats to *construct validity*, which concern our measurements of the constructs of interest (i.e., the phenomena underlying the independent and dependent variables),
- threats to *internal validity*, which concern our supposition of a causal relation between the phenomena underlying the independent and dependent variables, and
- threats to *external validity*, which concern our ability to generalize our results.

3.6.1 Construct Validity

Construct validity deals with the issue of whether or not we are measuring what we purport to be measuring. In these studies, our measurements for the rate of fault detection and the APFD values based on them are highly accurate, but APFD is not the only possible measure of rate of fault detection. For example, our measures assign no value to subsequent test cases that detect a fault already detected; such inputs may, however, help debuggers

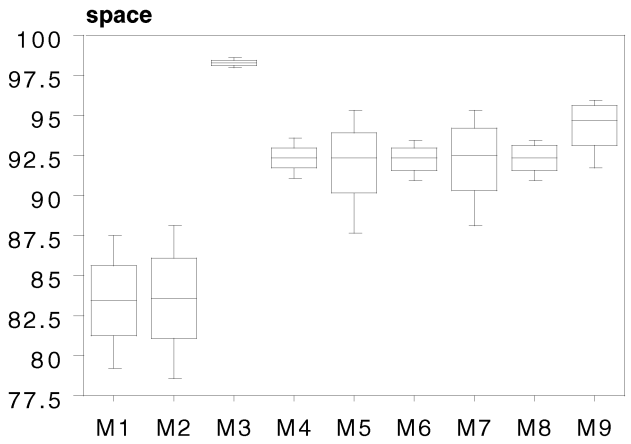


Fig. 7. APFD boxplot for Study 3. The techniques are: M_1: untreated, M_2: random, M_3: optimal, M_4: stmt-total, M_5: stmt-addtl, M_6: branch-total, M_7: branch-addtl, M_8: FEP-total, and M_9: FEP-addtl.

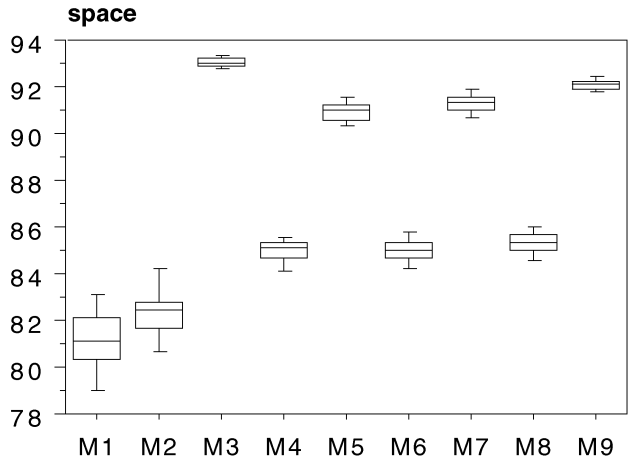


Fig. 8. APFD boxplot for Study 4. The techniques are: M_1: untreated, M_2: random, M_3: optimal, M_4: stmt-total, M_5: stmt-addtl, M_6: branch-total, M_7: branch-addtl, M_8: FEP-total, and M_9: FEP-addtl.

isolate the fault and for that reason might be worth measuring. Further, our APFD measures do not account for the possibility that faults and test cases may have different costs. One might not even want to measure the *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. Ultimately, because APFD only partially captures the aspects of the effectiveness of prioritization, we will need to consider other measures for purposes of assessing effectiveness.

Another threat to construct validity where FEP-based prioritization techniques are concerned involves our FEP 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. We use mutation analysis to provide an estimate of these FEP values; however, other estimates might be more precise and might increase the effectiveness of FEP-based techniques.

Finally, our empirical measurements have focused on measures of effectiveness of prioritization techniques, without measuring their cost. There are arguments in favor of this approach, as we have stated, and our worst-case analyses provide information useful in assessing relative costs; nevertheless, there may be other situations (e.g., version-specific prioritization) in which more explicit cost information would be helpful. Empirical studies of such costs would be useful for assessing cost-benefit trade-offs in those situations.

3.6.2 Internal Validity

Our greatest concern with respect to internal validity in these studies is instrumentation effects that can bias our results. One source of such effects are faults in the prioritization and APFD measurement tools used. To reduce the likelihood of such effects, we performed code reviews on all tools and validated tool outputs on a small but tractable program (130 lines, 12 versions, 15 test suites) and on several of the Siemens test suites.

Instrumentation effects can also be caused by differences in the test process inputs: the code to be tested, the locality of program changes, or the composition of the test suite. At this time, we do not control for effects related to types of test suites, nor for the structure of the subject programs or for the locality of program changes. To limit problems related to this, we applied each prioritization algorithm to each test suite and each subject program.

TABLE 6
Bonferroni Means Separation Tests for Study 3

space		
Grouping	Mean	Technique
A	98.2718	optimal
B	94.2316	FEP-addtl
C	92.3115	stmt-total
C	92.2944	FEP-total
C	92.2564	branch-total
C	92.0828	branch-addtl
C	91.8104	stmt-addtl
D	83.4450	random
D	83.3274	untreated
df= 7091 MSE= 6.0074 Critical Value of T= 3.20		
Minimum Significant Difference= 0.8011 (α=.05)		

TABLE 7
Bonferroni Means Separation Tests for Study 4

space		
Grouping	Mean	Technique
A	93.0357	optimal
B	92.0893	FEP-addtl
C	91.3140	branch-addtl
C	90.9310	stmt-addtl
D	85.2791	FEP-total
D	84.9914	branch-total
D	84.9898	stmt-total
E	82.2537	random
F	81.1423	untreated
df= 441 MSE= 0.726194 Critical Value of T= 3.22		
Minimum Significant Difference= 0.5484 (α=.05)		

3.6.3 External Validity

The threats to external validity of our studies are centered around the issue of how representative the subjects of our studies are.

The Siemens programs, although nontrivial, are small, and larger programs may be subject to different cost-benefit trade-offs. `Space` is a "real" program; however, it is only one such program.

Several threats to validity concern the faults and faulty versions of programs used in our study. The faults placed in the Siemens programs were synthetic (seeded). The faults in `space` were real faults, reportedly discovered during its development, but faults found in development may differ from faults found later in independent test; moreover, these faults represent only one set of faults found by one development team in one program. Our studies using mutants as faults let us consider a wider set of types of faults, and mutants do represent a class of faults that occur in practice, however, they represent only a certain class of such faults. Another source of threats involves our decision to use single-fault versions; this decision facilitated the experimentation and measurements but, in practice, faults may occur in many different distributions.

Other threats involve our test cases and test suites. Our branch-coverage-adequate test suites represent only one variety of test suites and, although they are constructed, for the Siemens programs from a mix of functional and code-coverage-based test cases, they may not represent a distribution of those test cases that would occur in practice. A similar statement applies to the test suites for `space` constructed from a mix of randomly generated and code-coverage-based test cases.

In general, however, such threats to external validity as these can be addressed only by additional studies on additional subjects. The studies in this paper begin this process and future work must continue it.

4 Additional Discussion and Practical Implications

Keeping the threats to validity for these empirical studies in mind, our data and analysis provide insights into the effectiveness of test case prioritization generally and into the relative effectiveness of the prioritization techniques that we examined. We now discuss these insights and their possible implications for the practical application of test case prioritization and for further research on prioritization.

Of greatest practical significance, our data and analysis indicates that test case prioritization can substantially improve the rate of fault detection of test suites. All of the heuristics that we examined produced such improvements overall and in only one case (`schedule` in Study 1) did test suites produced by any heuristic not outperform the untreated or randomly prioritized test suites. These results extended to the larger `space` program, as well.

Also, in our study, in almost every case—including on `space`—additional FEP prioritization outperformed prioritization techniques based on coverage overall. However, there were a few cases in which specific coverage-based techniques outperformed additional FEP prioritization and, in the cases in which additional FEP prioritization was the top performer, the total gain in APFD was not large. These results run contrary to our initial intuitions and suggest that, given their expense and depending upon the testing process used, additional FEP prioritization may not be as cost-effective as coverage-based techniques.

Again, considering overall results on the Siemens programs, branch-coverage-based techniques almost always performed as well as or better than their corresponding statement-coverage-based techniques. This suggested that, if cost factors for using such coverages are equal, branch-coverage-based techniques are a better choice. On `space`, however, this difference did not occur.

Considering differences between total and additional branch and statement-coverage-based techniques, there was no clear "winner" overall. Total coverage techniques did often outperform additional coverage techniques on the Siemens programs. Since the worst-case costs of total branch and statement coverage prioritization are a factor of test suite size less than the worst-case costs of additional branch and statement coverage prioritization; these results suggest that, in cases like these, the less expensive total-coverage prioritization schemes may be more cost-effective than additional-coverage schemes. This result did not extend fully to the larger and more realistic program `space`, however; in fact, on `space` in Study 3 additional and total coverage techniques were not significantly different and, in Study 4, additional coverage techniques outperformed total coverage techniques by a wide margin.

Initial investigation of `space` suggests that this difference may be due to the fact that a relatively high percentage of the original faults in `space` occur on relatively frequently executed ("mainline") paths, whereas the mutants for `space` are (by construction) relatively evenly distributed among code statements. On average, total coverage prioritization tends to reexecute mainline paths many times prior to attempting to reach more obscure paths. This increases the probability that total coverage prioritization will reveal faults on the mainline paths earlier in testing than will additional coverage prioritization, driving up the APFD score for total coverage prioritization.

Finally, in Study 1, we observed that randomly prioritized test suites typically outperformed untreated test suites. We conjecture that this difference is due to the type of test suites and faults used in Study 1. As described in Section 3.4, the test suites for the Siemens programs were generated by greedily selecting test cases from test pools. The order in which test cases were added to suites during this process constitutes their untreated order. We suspect that this process caused test cases added to the "ends" of the test suites to cover (on average) harder to reach statements than test cases added to the "beginnings" of the test suites. The faults provided with the Siemens programs are relatively hard to detect. A disproportionate number reside in harder-to-reach statements and are detected (more often) by test cases that are added later to the test suites than those added earlier. Random prioritization essentially redistributes test cases that reach and expose these faults throughout the test suites and, thus, could cause the faults to be detected more quickly.

The different results obtained in the other studies support this conjecture. The faults used in our second and

fourth studies are more widely distributed among program statements than the faults used in the first study, and the faults used in the second, third, and fourth studies include a mix of faults that are relatively easy to detect and faults that are relatively difficult to detect. Thus, we do not expect these faults to be detected more frequently by test cases at the ends of the untreated test suites such that randomizing the suites would redistribute the fault-revealing test cases more evenly.

These experiences with "untreated" test suites are noteworthy for the implications they raise for further empirical studies: Empiricists need to be aware of these implications in conducting such studies.

5 Conclusions and Future Work

In this paper, we have described several techniques for prioritizing test cases for regression testing and empirically examined their relative abilities to improve how quickly faults can be detected during regression testing. Our results suggest that these techniques can improve the rate of fault detection of test suites and that this result occurs even for the least sophisticated (and least expensive) techniques.

Our studies of FEP-based techniques, due to the expense of those techniques, at present are primarily of significance to researchers and further study of these techniques is necessary to determine whether they can be cost-effective. Our results with respect to code-coverage-based techniques, however, have immediate practical implications, suggesting that, if code coverage techniques are used in testing, they can be leveraged for additional gains through prioritization.

The results of our studies suggest several avenues for future work. First, we intend to perform additional studies using other programs and types of test suites, and a wider range and distribution of faults. Also, our APFD measure provides one benchmark with which to compare prioritization techniques, but other measures are possible, and we intend to empirically investigate some of those measures.

Second, because our analysis revealed a sizable performance gap between prioritization heuristics and optimal prioritization, and our FEP-based techniques did not bridge this gap, we are investigating alternative techniques. One such alternative involves a simpler use of mutation, in which we prioritize test cases based on their total mutation score, or the number of additional mutants they expose. Another possibility involves employing alternative estimates of fault-exposing potential, such as a measure based on static and dynamic dependencies in the code [12]. Prioritization techniques such as those we have investigated could also be extended to incorporate other forms of information. For instance, prioritization techniques could consider information on probabilities of modifications. Techniques that incorporate static measures of fault-proneness [22] may also be useful. Techniques that account for differing test case and fault costs could more accurately reflect practical trade-offs.

Third, differences in the performance of the various prioritization techniques we investigated, such as differences between the performance of total statement coverage and total branch coverage, mandate further study of the factors that underlie the relative effectiveness of various techniques. A desirable outcome of such a study would be procedures for predicting which prioritization techniques would be most effective for particular programs, types of test suites, and classes of modifications.

Finally, the test case prioritization problem, in general, has many more facets than we have investigated here. For example, we have considered only one possible prioritization objective; other objectives, such as those listed in Section 2, are also of interest. Further, we have examined only general prioritization techniques; version-specific techniques are also of interest. Processes that combine regression test selection and minimization with prioritization may be cost-effective. Moreover, our investigation has been confined to prioritization for regression testing, but it may also be beneficial to order tests during their initial creation, for use in the initial testing of software.

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

Acknowledgments

This work was supported in part by the US National Science Foundation Faculty Early Career Development Award CCR-9703108 presented to Oregon State University, and by the US National Science Foundation Award CCR-9707792 to Ohio State University and Oregon State University. Siemens Corporate Research supplied the Siemens programs. Alberto Pasquini, Phyllis Frankl, and Filip Vokolos provided the `Space` program and many of its test cases. Toto Sutarso assisted with the statistical analysis. Qiang Liu contributed to early discussions of the work. The anonymous reviewers provided suggestions that were both helpful and insightful. A preliminary version of this paper appeared in the Proceedings of the IEEE International Conference on Software Maintenance, August, 1999 [31].

References

[1] A. Avritzer and E.J. Weyuker, "The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software," *IEEE Trans. Software Eng.,* vol. 21, no. 9, pp. 705–716, Sept. 1995.

[2] M. Balcer, W. Hasling, and T. Ostrand, "Automatic Generation of Test Scripts from Formal Test Specifications," *Proc. Third Symp. Software Testing, Analysis, and Verification,* pp. 210–218, Dec. 1989.

[3] T. Ball, "On the Limit of Control Flow Analysis for Regression Test Selection," *Proc. ACM Int'l Symp. Software Testing and Analysis,* pp. 134–142, Mar. 1998.

[4] B. Beizer, *Software Testing Techniques.* New York: Van Nostrand Reinhold, 1990.

[5] D. Binkley, "Semantics Guided Regression Test Cost Reduction," *IEEE Trans. Software Eng.,* vol. 23, no. 8, pp. 498–516, Aug. 1997.

[6] T.Y. Chen and M.F. Lau, "Dividing Strategies for the Optimization of a Test Suite," *Information Processing Letters,* vol. 60, no. 3, pp. 135–141, Mar. 1996.

[7] Y.F. Chen, D.S. Rosenblum, and K.P. Vo, "TestTube: A System for Selective Regression Testing," *Proc. 16th Int'l Conf. Software Eng.,* pp. 211–222, May 1994.

[8] M.E. Delamaro and J.C. Maldonado, "Proteum—A Tool for the Assessment of Test Adequacy for C Programs," *Proc. Conf. Performability in Computing Systems (PCS '96),* pp. 79–95, July 1996.

[9] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer,* vol. 11, no. 4, pp. 34–41, Apr. 1978.

[10] R.J. Freund and R.C. Littell, *SAS for Linear Models: A Guide to the ANOVA and GLM Procedures.* Cary, N.C.: SAS Institute Inc., 1981.

[11] M.R. Garey and D.S. Johnson, *Computers and Intractability.* New York: W.H. Freeman, 1979.

[12] T. Goradia, "Dynamic Impact Analysis: A Cost-Effective Technique to Enforce Error-Propagation," *Proc. ACM Int'l Symp. Software Testing and Analysis,* pp. 171–181, June 1993.

[13] R.G. Hamlet, "Testing Programs with the Aid of a Compiler," *IEEE Trans. Software Eng.,* vol. 3, no. 4, pp. 279–290, July 1977.

[14] R.G. Hamlet, "Probable Correctness Theory," *Information Processing Letters,* vol. 25, pp. 17–25, Apr. 1987.

[15] M.J. Harrold, R. Gupta, and M.L. Soffa, "A Methodology for Controlling the Size of a Test Suite," *ACM Trans. Software Eng. and Methodology,* vol. 2, no. 3, pp. 270–285, July 1993.

[16] 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, The Ohio State Univ., Mar. 1997.

[17] J.T. Helwig, *SAS Introductory Guide.* SAS Institute, Inc., 1978.

[18] W.E. Howden, "Weak Mutation Testing and Completeness of Test Sets," *IEEE Trans. Software Eng.,* vol. 8, no. 7, pp. 371–379, July 1982.

[19] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria," *Proc. 16th Int'l Conf. Software Eng.,* pp. 191–200, May 1994.

[20] H.K.N. Leung and L. White, "Insights Into Regression Testing," *Proc. Conf. Software Maintenance,* pp. 60–69, Oct. 1989.

[21] H.K.N. Leung and L.J. White, "A Study of Integration Testing and Software Regression at the Integration Level," *Proc. Conf. Software Maintenance,* pp. 290–300, Nov. 1990.

[22] J.C. Munson and T.M. Khoshgoftaar, "The Detection of Fault-Prone Programs," *IEEE Trans. Software Eng.,* vol. 18, no. 5, pp. 423–433, May 1992.

[23] A.J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf, "An Experimental Determination of Sufficient Mutation Operators," *ACM Trans. Software Eng. and Methodology,* vol. 5, no. 2, pp. 99–118, Apr. 1996.

[24] K. Onoma, W-T. Tsai, M. Poonawala, and H. Suganuma, "Regression Testing in an Industrial Environment," *Comm. ACM,* vol. 41, no. 5, pp. 81–86, May 1988.

[25] T.J. Ostrand and M.J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests," *Comm. ACM,* vol. 31, no. 6, pp. 676–686, June 1988.

[26] L. Ott, *An Introduction to Statistical Methods and Data Analysis,* third ed. Boston, Mass.: PWS-Kent Publishing Company, 1988.

[27] S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Trans. Software Eng.,* vol. 11, no. 4, pp. 367–375, Apr. 1985.

[28] G. Rothermel and M.J. Harrold, "Analyzing Regression Test Selection Techniques," *IEEE Trans. Software Eng.,* vol. 22, no. 8, pp. 529–551, Aug. 1996.

[29] G. Rothermel and M.J. Harrold, "A Safe, Efficient Regression Test Selection Technique," *ACM Trans. Software Eng. and Methodology,* vol. 6, no. 2, pp. 173–210, Apr. 1997.

[30] 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," *Proc. Int'l Conf. Software Maintenance,* pp. 34–43, Nov. 1998.

[31] G. Rothermel, R.H. Untch, C. Chu, and M.J. Harrold, "Test Case Prioritization: An Empirical Study," *Proc. Int'l Conf. Software Maintenance,* pp. 179–188, Aug. 1999.

[32] M.C. Thompson, D.J. Richardson, and L.A. Clarke, "An Information Flow Model of Fault Detection," *Proc. ACM Int'l Symp. Software Testing and Analysis,* pp. 182–192, June 1993.

[33] J. Voas, "PIE: A Dynamic Failure-Based Technique," *IEEE Trans. Software Eng.,* vol. 18, no. 8, pp. 717–727, Aug. 1992.

[34] F.I. Vokolos and P.G. Frankl, "Pythia: A Regression Test Selection Tool Based on Textual Differencing," *Proc. Third Int'l Conf. Reliability, Quality & Safety of Software-Intensive Systems (ENCRESS '97),* May 1997.

[35] F.I. Vokolos and P.G. Frankl, "Empirical Evaluation of the Textual Differencing Regression Testing Technique," *Proc. Int'l Conf. Software Maintenance,* pp. 44–53, Nov. 1998.

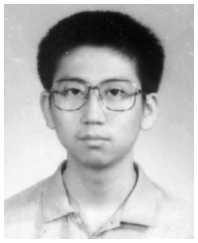
[36] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal, "A Study of Effective Regression Testing in Practice," *Proc. Eighth Int'l Symp. Software Reliability Eng.,* pp. 230–238, Nov. 1997.

[37] W.E. Wong, J.R. Horgan, S. London, and A.P. Mathur, "Effect of Test Set Minimization on Fault Detection Effectiveness," *Software–Practice and Experience,* vol. 28, no. 4, pp. 347–369, Apr. 1998.

[38] 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," *Proc. 21st Ann. Int'l Computer Software & Applications Conf.,* pp. 522–528, Aug. 1997.

Gregg Rothermel received the BA degree in philosophy from Reed College, Portland, Oregan, the MS degree in computer science from State University of New York, Albany, and the PhD degree in computer science from Clemson University, Clemson, South Carolina. He is currently an associate professor in the Computer Science Department at Oregon State University. His research interests include software engineering and program analysis, with emphases on the application of program analysis techniques to problems in software maintenance and testing, and on empirical studies. Previously he was the Vice President of Quality Assurance and Quality Control, Palette Systems, Incorporated. He is a recipient of the US National Science Foundation's Faculty Early Career Development Award and of the Oregon State University College of Engineering's Engelbrecht Young Faculty Award. He has served on the program committees for the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis, the 2000 International Conference on Software Engineering, the 2001 International Conference on Software Engineering, the SIGSOFT 2000 Eighth International Symposium the Foundations of Software Engineering (FSE-8), and the 2000 International Conference on Software Maintenance. He is a member of the IEEE Computer Society, the ACM, the ACM SIGSOFT, and the ACM SIGPLAN.

Roland H. Untch received the BA degree in management from Mundelein College (Loyola University Chicago), the MS degree in computer science from DePaul University, Chicago, Illinios, and the PhD degree in computer science from Clemson University, Clemson, South Carolina. He is an associate professor in the Department of Computer Science at Middle Tennessee State University. His research interests include software engineering, software testing, program analysis, and compilers. He is a member of the IEEE Computer Society and the ACM.



Chengyun Chu received the BS degree in computer science from Xi'an Jiaotong University, China, and the MS degree in computer science from Oregon State University. From 1998 to 1999, he worked as a research assistant in the Department of Computer Science at Oregon State University. He is currently working for Microsoft, Inc.

Mary Jean Harrold received the MS and PhD degrees in computer science from the University of Pittsburgh and the MS and MA degrees in mathematics from Marshall University. She is currently an associate professor in the College of Computing at the Georgia Institute of Technology where she leads the Aristotle Software Engineering Research Group. Her research interests include software engineering, with an emphasis on program analysis techniques that facilitate testing and maintenance tasks. She is a recipient of the US National Science Foundation's National Young Investigator Award. She served as program cochair of the 1997 International Conference on Software Maintenance. She served as program chair for the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis, as program cochair for the 2001 International Conference on Software Engineering, and as an associate editor for *IEEE Transactions on Software Engineering*. She is a member of the Computing Research Association's Committee on the Status of Women in Computing and serves as program director for the Computing Research Association's Distributed Mentor Project. She is a member of the IEEE Computer Society and the ACM.

▷ **For more information on this or any computing topic, please visit our Digital Library at** http://computer.org/publications/dlib.