University of Nebraska-Lincoln, Computer Science and Engineering
Technical Report # TR-UNL-CSE-2000-0005
Also presented in International Symposium of Software Testing and Analysis, 102-112, August 2000.

Prioritizing Test Cases for Regression Testing

Sebastian Elbaum
Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, Nebraska
elbaum@cse.unl.edu

Alexey G. Malishevsky
Computer Science Department
Oregon State University
Corvallis, Oregon
malishal@cs.orst.edu

Gregg Rothermel
Computer Science Department
Oregon State University
Corvallis, Oregon
grother@cs.orst.edu

ABSTRACT

Test case prioritization techniques schedule test cases in an order that increases their effectiveness in meeting some performance goal. One performance goal, *rate of fault detection*, is a measure of how quickly faults are detected within the testing process; an improved rate of fault detection can provide faster feedback on the system under test, and let software engineers begin locating and correcting faults earlier than might otherwise be possible. In previous work, we reported the results of studies that showed that prioritization techniques can significantly improve rate of fault detection. Those studies, however, raised several additional questions: (1) can prioritization techniques be effective when aimed at specific modified versions; (2) what tradeoffs exist between fine granularity and coarse granularity prioritization techniques; (3) can the incorporation of measures of fault proneness into prioritization techniques improve their effectiveness? This paper reports the results of new experiments addressing these questions.

1. INTRODUCTION

Software engineers often save the test suites they develop so that they can reuse those test suites later as their software evolves. Such test suite reuse, in the form of *regression testing*, is pervasive in the software industry [22]. 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. In such cases, testers may want to order their test cases so that those with the highest priority, according to some criterion, are run earlier than those with lower priority.

Test case prioritization techniques [24, 28] schedule test cases for regression testing in an order that increases their effectiveness at meeting some performance goal. For example, test cases might be scheduled in an order that achieves code coverage at the fastest rate possible, exercises features in order of expected frequency of use, or exercises subsystems in an order that reflects their past failure rates.

One potential goal of test case prioritization is that of increasing a test suite's *rate of fault detection* – a measure of how quickly that test suite detects faults during the testing process. An increased rate of fault detection can provide earlier feedback on the system under regression test and let developers begin locating and correcting faults earlier than might otherwise be possible. Such feedback can also provide earlier evidence that quality goals have not been met, allowing earlier strategic decisions about release schedules. Further, an improved rate of fault detection can increase the likelihood that if testing is prematurely halted, those test cases that offer the greatest fault detection ability in the available testing time will have been executed.

In previous work [24] we presented several techniques for prioritizing test cases, and empirically evaluated their abilities to improve rate of fault detection. Our results indicated that several of the techniques could improve rate of fault detection, and that this improvement could occur even for the least sophisticated (and least expensive) techniques.

Our results also raised several additional questions. First, we examined only "general prioritization", which attempts to select a test case order that will be effective *on average* over a succession of subsequent versions of the software. In regression testing, we are concerned with a particular version of the software, and we wish to prioritize test cases in a manner that will be most effective for that version. In this context, we are interested in "version-specific prioritization", and we are interested in the effectiveness of this prioritization relative to versions that contain multiple faults.

Second, the techniques we examined all operated at relatively *fine granularity* – that is, they involved instrumentation, analysis, and prioritization at the level of source code statements. For large software systems, or systems in which instrumentation at the statement level is not feasible, such techniques may not be sufficiently efficient. An alternative is to operate at a relatively *coarse granularity*; for example, at the function level. We expect, however, that coarse granularity techniques will not be as effective as fine granularity techniques. We wish to examine the cost-benefits tradeoffs that hold, for test case prioritization, across granularities.

Third, our analysis revealed a sizable performance gap between the results achieved by the prioritization techniques that we examined, and the optimal results achievable. We wish to at least partially bridge this gap, and we conjectured that by incorporating measures of fault proneness (e.g. [10, 21]) into our techniques we might be able to do so.

To investigate these questions we have performed new experiments; this paper reports their results. In the next section, we describe the test case prioritization problem and several issues relevant to its solution. Section 3 describes the test case prioritization techniques that we have studied. Section 4 describes our empirical studies, presenting research questions, experiment design, results and analysis. Section 5 discusses practical implications of those results. Section 6 presents conclusions and directions for future research.

2. TEST CASE PRIORITIZATION

We define the test case prioritization problem as follows:

> *The Test Case Prioritization Problem:*
>
> *Given*: T, a test suite; PT, the set of permutations of T; 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 the definition assumes that higher award values are preferable to lower ones.)

There are many possible goals for prioritization, for example:

- Testers may wish to increase the rate of fault detection of test suites – that is, the likelihood of revealing faults earlier in a run of regression tests using those suites.
- Testers may wish to increase the coverage of code in the system under test at a faster rate, 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 likelihood of revealing faults related to specific code changes earlier in the testing process.

These goals are stated qualitatively. To measure the success of a prioritization technique in meeting any such goal we must describe the goal quantitatively. In the definition of the test case prioritization problem, f represents such a quantification. In this work, we focus on the first of the goals just stated: increasing the likelihood of revealing faults earlier in the testing process. We describe this goal, informally, as one of improving our test suite's *rate of fault detection*: we provide a quantitative measure for this goal in Section 4.1.

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 [12]. 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, test case prioritization techniques must be heuristics.

We 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. Our hope is that the resulting prioritized suite will be more successful than the original suite at meeting the goal of the prioritization, *on average* over those subsequent releases.

In contrast, in *version-specific test case prioritization*, given program P and test suite T, 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'. 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.

Finally, in this paper we address the problem of prioritizing test cases for regression testing; however, test case prioritization can also be employed in the initial testing of software (see e.g. [2]). An important 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; such information is not available during initial testing.

3. PRIORITIZATION TECHNIQUES

Given any prioritization goal, various *test case prioritization techniques* may be utilized to meet that goal. For example, 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 have tended to fail in the past. 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 work we consider 14 test case prioritization techniques, which we classify into three groups. Table 1 lists these techniques by group. The first group is the *control* group, containing two "techniques" that serve as experimental controls. The second group is the *statement level* group, containing four fine granularity techniques; these techniques

Label	*Mnemonic*	*Description*
T1	random	randomized ordering
T2	optimal	ordered to optimize rate of fault detection
T3	st-total	prioritize on coverage of statements
T4	st-addtl	prioritize on coverage of statements not yet covered
T5	st-fep-total	prioritize on probability of exposing faults
T6	st-fep-addtl	prioritize on probability of faults, adjusted to consider previous test cases
T7	fn-total	prioritize on coverage of functions
T8	fn-addtl	prioritize on coverage of functions not yet covered
T9	fn-fep-total	prioritize on probability of exposing faults
T10	fn-fep-addtl	prioritize on probability of faults, adjusted to consider previous test cases
T11	fn-fi-total	prioritize on probability of fault existence
T12	fn-fi-addtl	prioritize on probability of fault existence, adjusted to consider previous test cases
T13	fn-fi-fep-total	prioritize on combined probabilities of fault existence and fault exposure
T14	fn-fi-fep-addtl	prioritize on combined probabilities of fault existence/exposure, adjusted on previous coverage

Table 1: Test case prioritization techniques considered.

were used in our earlier study [24] but here they are examined in the context of version-specific prioritization. The third group is the *function level* group, containing eight coarse granularity techniques; four are comparable to statement level techniques, and four add information on the probability of fault existence not utilized by the statement level techniques. Next, we briefly describe each technique.

3.1 Control techniques

T1: Random ordering.

As an experimental control, one prioritization "technique" that we consider is the random ordering of the test cases in the test suite.

T2: Optimal ordering.

As a second experimental control, we consider an optimal ordering of the test cases in the test suite. We can obtain such an ordering in our experiments because we utilize programs with known faults and can determine which faults each test case exposes: this lets us determine the ordering of test cases that maximizes a test suite's rate of fault detection. In practice, of course, this is not a practical technique, but it provides an upper bound on the effectiveness of the other heuristics that we consider.

3.2 Statement level techniques

T3: Total statement coverage prioritization.

By instrumenting a program we can determine, for any test case, the number of statements in that program that were exercised by that test case. We can prioritize these test cases according to the total number of statements they cover simply by sorting them in order of total statement coverage achieved.

T4: 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 covering statements that have not yet been covered. Additional statement coverage prioritization greedily selects a test case that yields the greatest statement coverage, then adjusts the coverage data about subsequent test cases to indicate their coverage of statements not yet covered, and then repeats this process, until all statements covered by at least one test case have been covered. When all statements have been covered, remaining test cases must also be ordered; we do this (recursively) by resetting all statements to "not covered" and reapplying additional statement coverage on the remaining test cases.

T5: Total FEP prioritization.

The ability of a fault to be exposed by a test case depends not only on whether the test case executes a faulty component, but also on the probability that a fault in that statement will cause a failure for that test case [13, 15, 25, 26]. Although any practical determination of this probability must be an approximation, we wish to know whether the use of such an approximation might yield a prioritization technique superior in terms of rate of fault detection than techniques based solely on code coverage.

To approximate the fault-exposing-potential (FEP) of a test case we used mutation analysis [7, 14]. Given program P and test suite T, for each test case $t \in T$, for each statement s in P, we determined the mutation score $ms(s,t)$ of t on s to be the ratio of mutants of s exposed by t to total mutants of s. We then calculated, for each test case t_k in T, an *award value* for t_k, by summing all $ms(s,t_k)$ values. Total fault-exposing-potential prioritization orders the test cases in a test suite in order of these award values.

Given this approximation method, FEP prioritization is more expensive than code-coverage-based techniques due to the expense of mutation analysis. If FEP prioritization shows promise, however, this would motivate a search for cost-effective approximators of fault-exposing potential.

T6: Additional FEP prioritization.

Analogous to the extensions made to total statement coverage prioritization to yield additional statement coverage prioritization, we extend total FEP prioritization to create additional fault-exposing-potential (FEP) prioritization. In additional FEP prioritization, after selecting a test case t, we lower the award values for all other test cases that exercise statements exercised by t to reflect our increased confidence in the correctness of those statements; we then select a next test case, repeating this process until all test cases have been ordered. This approach lets us account for the fact that additional executions of a statement may be less valuable than initial executions.

3.3 Function level techniques

T7: Total function coverage prioritization.

Analogous to total statement coverage prioritization but operating at the level of functions, this technique prioritizes test cases according to the total number of functions they execute.

T8: Additional function coverage prioritization.

Analogous to additional statement coverage prioritization but operating at the level of functions, this technique prioritizes test cases (greedily) according to the total number of additional functions they cover.

T9: Total FEP (function level) prioritization.

This technique is analogous to total FEP prioritization at the statement level. To translate that technique to the function level, we required a function level approximation of fault-exposing potential. We again used mutation analysis, computing, for each test case t and each function f, the ratio of mutants in f exposed by t to mutants of f executed by t. Summing these values we obtain award values for test cases. We then apply the same prioritization algorithm as for total FEP (statement level) prioritization, substituting functions for statements.

T10: Additional FEP (function level) prioritization.

This technique extends the total FEP (function level) technique in the same manner in which we extended the total FEP (statement level) technique.

T11: Total fault index (FI) prioritization.

Faults are not equally likely to exist in each function; rather, certain functions are more likely to contain faults than others. This fault proneness can be associated with measurable software attributes [1, 3, 5, 18, 19]. We attempt to take advantage of this association by prioritizing test cases based on their history of executing fault prone functions.

To represent fault proneness, we use a *fault index* based on principal component analysis [10, 21].[1] Generating fault indexes requires measurement of each function in the new version, generation of fault indexes for the new version, and comparison of the new indexes against the indexes calculated for the baseline version. Each function is thereby assigned an absolute fault index representing the fault proneness for that function, based on the complexity of the changes that were introduced into that function.

Given these fault indexes, total fault index coverage prioritization is performed in a manner similar to total function coverage. For each test case, we compute the sum of the fault indexes for every function that test case executes. Then, we sort those test cases in decreasing order of these sums.

T12: Additional fault-index (FI) prioritization.

Additional fault index coverage prioritization is accomplished in a manner similar to additional function coverage. The set of functions that have been covered by previously executed test cases is maintained. If this set contains all functions (more precisely, if no test case adds anything to this coverage), the set is reinitialized to $\emptyset$. To find the next best test case we compute, for each test case, the sum of the fault indexes for each function that test case executes, except for functions in the set of covered functions. The test case for which this sum is the greatest wins. This process is repeated until all test cases have been prioritized.

T13: Total FI with FEP coverage prioritization.

We hypothesized that, by utilizing *both* an estimate of fault exposing potential *and* an estimate of fault proneness, we might be able to achieve a superior rate of fault detection. Therefore, in this technique, we first apply total fault index prioritization to all test cases; then, for all test cases that possess equal fault index award values, we apply total FEP prioritization as a secondary ordering.

T14: Additional FI with FEP coverage prioritization.

We extend the previous technique to an "additional" variant. In this technique, we use additional fault index prioritization to obtain an initial test case ordering; we then apply FEP prioritization to rank all test cases possessing equal fault-index-based award values.

4. THE EXPERIMENTS

We are interested in the following research questions.

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

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

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

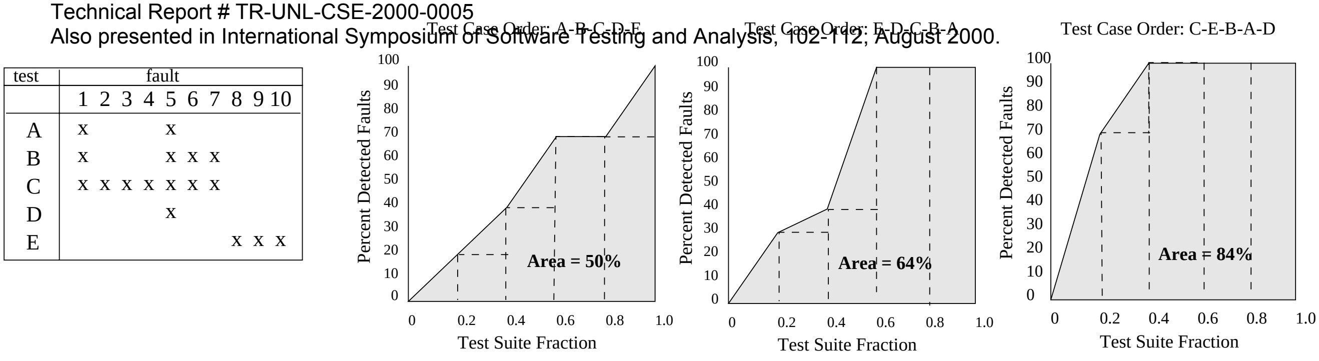
4.1 Efficacy and APFD Measures

To quantify the goal of increasing a test suite's rate of fault detection, we use a weighted average of the percentage of faults detected, or *APFD*, over the life of the suite. These values range from 0 to 100; higher APFD numbers mean faster (better) fault detection rates.

For illustration, consider a program with 10 faulty versions and a test suite of 5 test cases, **A** through **E**. Figure 1.A shows the fault detecting ability of these test cases.

Suppose we place the test cases in order **A**–**B**–**C**–**D**–**E** to form a prioritized test suite $T1$. Figure 1.B shows the percentage of detected faults versus the fraction of the test suite $T1$ used. After running test case **A**, 2 of the 10 faults are detected; thus 20% of the faults have been detected after 0.2 of test suite $T1$ has been used. After running test case **B**, 2 more faults are detected and thus 40% of the faults have been detected after 0.4 of the test suite has been used. In Figure 1.B, the area inside the inscribed rectangles (dashed boxes) represents the weighted percentage of faults detected over the corresponding fraction of the test suite. The solid lines connecting the corners of the inscribed rectangles interpolate the gain in the percentage of detected faults. This

[1] Due to space limitations we do not describe the mechanisms of the method, but details are given in [10].



test	fault									
	1	2	3	4	5	6	7	8	9	10
A	x				x					
B	x				x	x	x			
C	x	x	x	x	x	x	x			
D					x					
E								x	x	x

A. Test suite and faults exposed **B. APFD for prioritized suite T1** **C. APFD for prioritized suite T2** **D. APFD for prioritized suite T3**

Figure 1: Example illustrating the APFD measure.

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

Table 2: Experiment subjects.

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 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% in this example.

Figure 1.C reflects what happens when the order of test cases is changed to **E**–**D**–**C**–**B**–**A**, yielding a "faster detecting" suite than $T1$ with APFD 64%. Figure 1.D 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 APFD 84%.

4.2 Experiment Instrumentation

4.2.1 Programs

We used eight C programs as subjects. 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 [17]. We refer to these as the *Siemens programs*. The eighth program, `space`, is a program developed for the European Space Agency. We refer to this program as the *Space program*. Table 2 describes the programs.

Siemens programs.

The Siemens programs perform various tasks: `tcas` is an aircraft collision avoidance system, `schedule2` and `schedule` are priority schedulers, `tot_info` computes statistics, `print_tokens` and `print_tokens2` are lexical analyzers, `replace` performs pattern matching and substitution. For each program, the Siemens researchers created a *test pool* of black-box test cases using the category partition method and TSL tool [4, 23]. They then augmented this test pool with manually created white-box test cases to ensure that each exercisable statement, edge, and definition-use pair in the base program or its control flow graph was exercised by at least 30 test cases. The researchers also created faulty versions of each program by modifying code in the base version; in most cases they modified a single line of code, and in a few cases they modified between 2 and 5 lines of code. Their goal was to introduce faults that were as realistic as possible, based on their experience with real programs. To obtain meaningful results, the researchers retained only faults that were detectable by at least 3 and at most 350 test cases in the associated test pool.

Space program.

The Space program is an interpreter for an array definition language (ADL). The program reads a file of ADL statements, and checks the contents of the file for adherence to the ADL grammar and specific consistency rules. If the ADL file is correct, the Space program outputs an array data file containing a list of array elements, positions, and excitations; otherwise the program outputs error messages. The Space program has 35 versions, each containing a single fault: 30 of these were discovered during the program's development, 5 more were discovered by ourselves. The test pool for the Space program was constructed in two phases. We began with a pool of 10,000 randomly generated test cases created by Vokolos and Frankl [27]. Then we added new test cases until every executable edge in the program's control flow graph was exercised by at least 30 test cases. This process yielded a test pool of 13,585 test cases.

Test Suites.

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 1000 branch-coverage-adequate test suites for each program. For our experimentation, we randomly selected 50 of these test suites for each program.

Versions.

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

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

4.2.2 Prioritization and analysis tools

To perform the experiments we required several tools. Our test coverage and control-flow graph information was provided by the Aristotle program analysis system [16]. We created prioritization tools implementing the techniques outlined in Section 3. To obtain mutation scores for use in FEP prioritization we used the Proteum mutation system [6]. To obtain fault index information we used three tools [9, 11]: source code measurement tools for generating complexity metrics, a fault index generator, and a comparator for evaluating each version against the baseline version.

4.3 Experiments: Design and Results

To address our research questions, we designed a family of experiments. Each experiment included five stages: (1) stating the research question in terms of an hypothesis, (2) formalizing the experiment through a robust design, (3) collecting data, (4) analyzing data to test the hypothesis, and (5) identifying the threats to the experiment's validity. In general, each experiment examined the results of applying certain test case prioritization techniques to each program and its set of versions and test suites.

To provide an overview of all the collected data[3] we include Figure 2 with box plots.[4] The figure contains separate plots for an "all program" total (bottom) and for each of the programs. Each plot contains a box showing the distribution of APFD scores for each of the 14 techniques. See Table 1 for a legend of the techniques.

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

4.3.1 RQ1: Version-specific prioritization

Our first research question considers whether version-specific test case prioritization can improve the fault-detection abilities of test suites. We conjectured that differences in granularity would cause significant differences in fault detection, so we designed two experiments to respond to this question: Experiment 1a involving statement level techniques and Experiment 1b involving function level techniques. This separation into two experiments gave us more power to determine differences among the techniques within each group. Both experiments followed the same factorial design: all combinations of all levels of all factors were investigated. The factors were program and prioritization technique. Within programs, there were 8 levels with 29 versions and 50 test suites of different size per level. Within techniques, there were 4 levels in each experiment. Experiment 1a examined st-total, st-addtl, st-fep-total and st-fep-addtl. Experiment 1b examined fn-total, fn-addtl, fn-fep-total and fn-fep-addtl.

Observe that in [24], optimal and random techniques were used as control groups, and it was determined that they were significantly different from a given set of statement level techniques. In these two experiments, we elected to exclude optimal and random to focus on differences between actual techniques at each level of granularity. (To provide a frame of reference for all the presented techniques, optimal and random are presented in Section 4.3.4.)

For Experiments 1a and 1b we performed ANOVA analyses considering main effects and interaction among the factors. The top half of Table 3 presents results for Experiment 1a, considering all programs. The results indicate that there is enough statistical evidence to reject the null hypothesis, that is, the means for the APFD values generated by different statement level techniques were different. However, the analysis also indicates that there is significant interaction between techniques and programs.[5] The difference in response between techniques is not the same for all programs. Thus, individual and careful interpretation is necessary.

Source	d. f.	M. S.	F	P > F
Model	31	146227.87	397.72	0.0001
Error	31836	367.66		
Bonferroni Mininum Significant Difference: 0.80				

Grouping	Mean	Technique
A	78.88	st-fep-addtl
B	76.99	st-fep-total
B	76.30	st-total
C	74.44	st-addtl

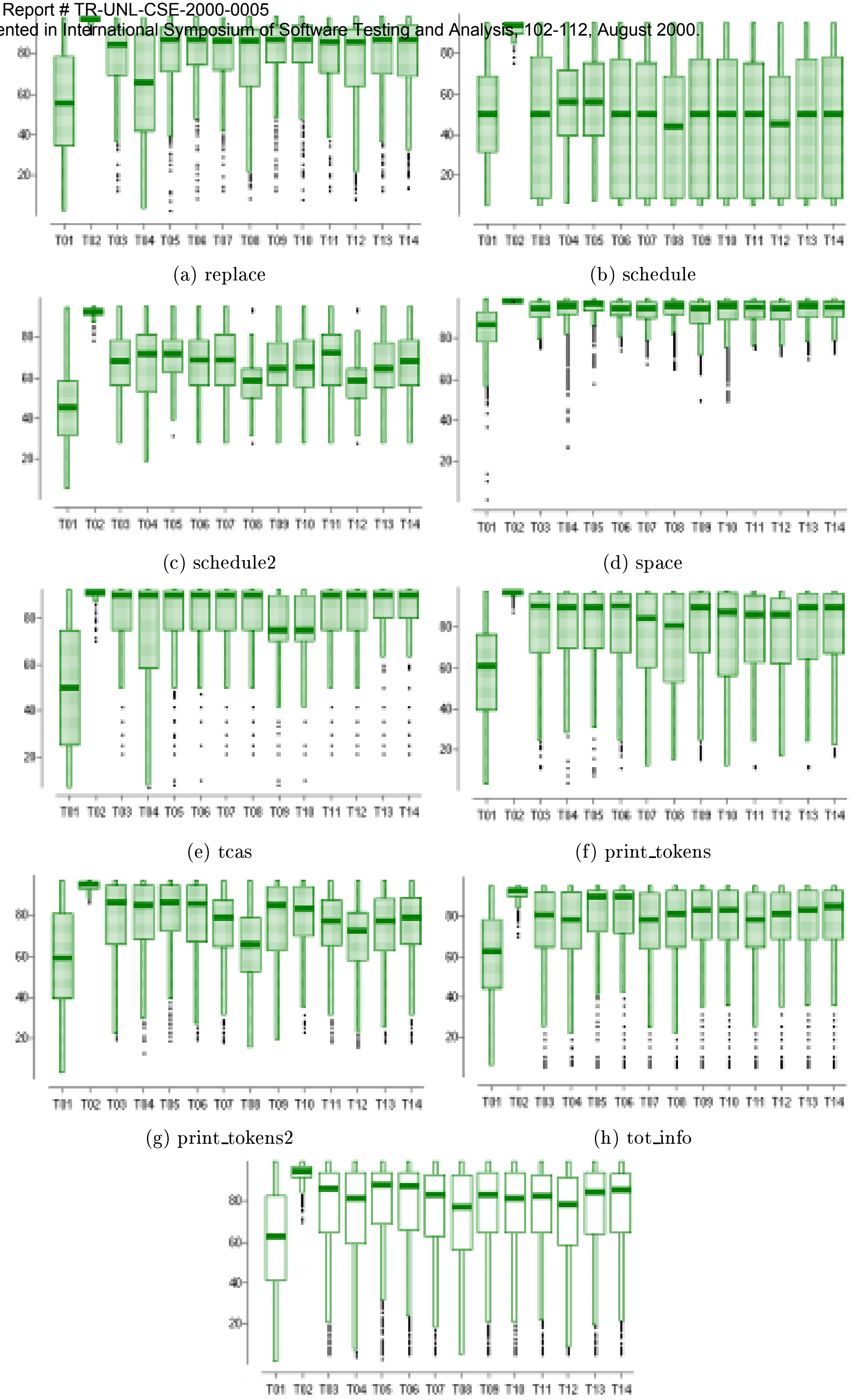
Table 3: ANOVA analysis and Bonferroni means separation tests, statement level techniques, all programs.

[2] The number of versions, 29, constitutes the minimum among the maximum number of versions that could be generated for each program given the interference constraints.

[3] To conserve space, data belonging to separate experiments have been presented together.

[4] Box plots 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.

[5] We present only a subset of the ANOVA analysis for all programs (note that the interaction values are not present), and we do not present individual ANOVA results for each program. However, individual results for each program are available in [8].



(a) replace

(b) schedule

(c) schedule2

(d) space

(e) tcas

(f) print_tokens

(g) print_tokens2

(h) tot_info

(i) all programs

Figure 2: APFD boxplots, all programs. The horizontal axes list techniques (see Table 1 for a legend of the techniques), and the vertical axes list APFD scores.

The APFD means ranked the techniques, the ANOVA analysis evaluated whether the techniques differed, a multiple comparison procedure using Bonferroni analysis quantifies how the techniques differ from each other. The bottom half of Table 3 presents results for the statement level techniques. Techniques with the same grouping letter are not significantly different. For example, st-fep-total has a larger mean than st-total but they are grouped together because they are not significantly different. On the other hand, the st-fep-addtl technique, which uses FEP information and additional coverage, is significantly better than the other techniques.

Table 4 presents analogous results for the ANOVA and Bonferroni analyses for Experiment 1b. The interaction effects between techniques and programs were significant for these function-level techniques. The results also show significant differences among the techniques. Moreover, the techniques ranked in the same order as their statement-level equivalents, with fn-fep-addtl first, fn-fep-total second, fn-total third, and fn-addtl last. However, in this case, the top three techniques were not significantly different from each other. At a minimum, this result suggests that our method for estimating FEP values at the function level may not be as powerful as our method for estimating those values at the statement level; further study is needed to determine whether this result generalizes, and whether more effective function-level estimators can be found.

Source	d. f.	M. S.	F	P > F
Model	31	169080.15	436.93	0.0001
Error	31836	386.96		
Bonferroni Minimum Significant Difference:				0.82

Grouping	Mean	Technique
A	75.59	fn-fep-addtl
A	75.48	fn-fep-total
A	75.09	fn-total
B	71.66	fn-addtl

Table 4: ANOVA analysis and Bonferroni means separation tests, basic function level techniques, all programs.

4.3.2 *RQ2: Granularity effects*

Our second research question concerns the relation between fine and coarse granularity prioritization techniques. Initial observations on the data led us to hypothesize that granularity has an effect on APFD values. This is evident in the boxplots, where for all cases, the mean APFD values for function level techniques were smaller than the APFD values for corresponding statement level techniques. For example, the mean APFD for fn-fep-addtl was 75.59, but for st-fep-addtl it was 78.88. The radar chart in Figure 3 further confirms this observation. In the radar chart, each technique has its own APFD value axis radiating from the center point. There are two polygons representing the granularities at the statement and at the function level. The radar chart shows that each function level technique has a smaller APFD than each statement level technique, and that statement level techniques as a whole are better (they cover a larger surface) than function level techniques.

To address this research question we performed an experiment (Experiment 2), similar to those performed to address RQ1: we used the same experiment design, but performed pairwise comparisons among the following pairs of techniques: (st-total,fn-total), (st-addtl,fn-addtl), (st-fep-total,fn-fep-total), and (st-fep-addtl,fn-fep-addtl).

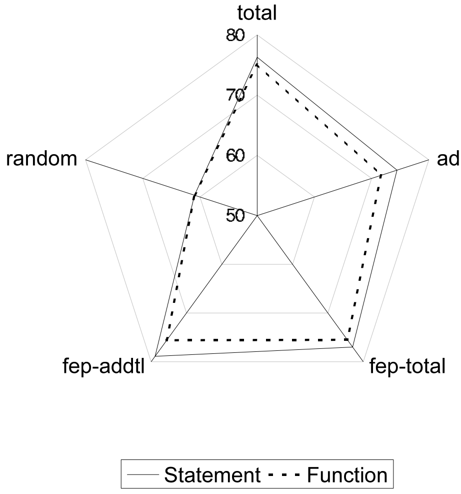


Figure 3: Radar chart.

The four orthogonal contrasts were significantly different under a Student Multiple t test. That is, for these four pairs of techniques, different levels of granularity had a major effect on the value of the fault detection rate. Thus, in spite of the different rankings in Experiments 1a and 1b, there is enough statistical evidence to confirm that statement level techniques are more effective that function level techniques.

4.3.3 *RQ3: Adding prediction of fault proneness*

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

The ANOVA analysis of the data collected in this experiment (see Table 5) indicated that these techniques were significantly different. We then followed the same procedure used earlier, employing a Bonferroni analysis to gain insight into the differences. The results were not what we expected. Although fn-fi-fep-addtl had the largest APFD mean, it was not significantly different from fn-fep-addtl. That means that the combination of FEP and fault proneness measures did not increment the techniques' efficiencies as measured by APFD. The lack of significant difference also held in other cases where fault-proness estimation was added to the prioritization technique: fn-fi-fep-total and fn-fep-total, fn-fi-total and fn-total, and fn-fi-addtl and fn-addtl. This suggests that the fault-proneness and FEP estimators we employed did not significantly improve the power of our prioritization techniques.

Source	d. f.	M. S.	F	P > F
Model	63	166395.24	440.71	0.0001
Error	63736	377.59		
Bonferroni Minimum Significant Difference:				0.96

Grouping		Mean	Technique
A		76.34	fn-fi-fep-addtl
A	B	75.92	fn-fi-fep-total
A	B	75.63	fn-fi-total
A	B	75.59	fn-fep-addtl
A	B	75.49	fn-fep-total
	B	75.09	fn-total
C		72.62	fn-fi-addtl
C		71.66	fn-addtl

Table 5: ANOVA analysis and Bonferroni means separation tests, all function level techniques, all programs.

Source	d. f.	M. S.	F	P > F
Model	111	190818.46	529.84	0.0001
Error	111426	360.14		
Bonferroni Minimum Significant Difference:				1.039

Grouping		Mean	Technique
A		94.24	optimal
B		78.88	st-fep-addtl
	C	76.99	st-fep-total
D	C	76.34	fn-fi-fep-addtl
D	C	76.30	st-total
D	E	75.92	fn-fi-fep-total
D	E	75.63	fn-fi-total
D	E	75.59	fn-fep-addtl
D	E	75.49	fn-fep-total
F	E	75.09	fn-total
F		74.44	st-addtl
G		72.62	fn-fi-addtl
G		71.66	fn-addtl
H		59.73	random

Table 6: ANOVA analysis and Bonferroni means separation tests, all techniques, all programs.

These results contradict our expectations and results of previously published studies [20]. One possible source of this difference is the fault distribution within our subject programs: in several cases our program versions contain only single faults involving single code changes. In such cases, a fault index is diminished to a binary condition that indicates whether a function has changed or not, and its ability to act as an effective fault proneness indicator may be lessened. Although this conjecture requires further empirical study, one implication is that the relative usefulness of prioritization techniques for regression testing will vary with characteristics of the modified program, and for practical purposes, methods for predicting which techniques will be appropriate in particular situations should be sought.

4.3.4 Overall analysis

Finally, to gain an overall perspective on all techniques, we performed ANOVA and Bonferroni analyses on all the techniques including optimal and random (see Table 6). As expected, the ANOVA analysis showed significant differences among the techniques and the Bonferroni analysis generated groups which confirmed our previous observations. The most obvious observation is that the optimal technique was still significantly better than all other techniques; this suggests that there is still room for improvement in prioritization techniques. However, all techniques outperform random ordering. Another interesting observation is that some of the advanced function level techniques outperformed some statement-level techniques.

4.4 Threats to Validity

In this section present a synthesis of the potential threats to validity of our study, including: (1) threats to internal validity (could other effects on our dependent variables be responsible for our results); (2) threats to external validity (to what extent to our results generalize); (3) threats to construct validity (are our independent variables appropriate).

4.4.1 Threats to internal validity

(1) Faults in the prioritization and APFD measurement tools. To control for this threat, we performed code reviews on all tools, and validated tool outputs on a small but non-trivial program. (2) Differences in the code to be tested, the locality of program changes, and the composition of the test suite. To reduce this threat, we used a factorial design to apply each prioritization technique to each test suite and each subject program. (3) FEP and FI 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. Similar reasoning applies to our calculations of fault index values.

4.4.2 Threats to external validity

(1) Subject program representativeness. The subject programs are of small and medium size, and have simple fault patterns that we have manipulated to produce versions with multiple faults. Complex industrial programs with different characteristics may be subject to different cost-benefit tradeoffs. (2) Testing process representativeness. If the testing process we used is not representative of industrial ones, the results might be invalid. Control for these two threats can be achieved only through additional studies using a greater range of software artifacts.

4.4.3 Threats to construct validity

(1) 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 isolate the fault, and for that reason might be worth measuring. (2) APFD measures do not account for the possibility that faults and test cases may have different costs. (3) APFD only partially captures the aspects of the effectiveness of prioritization, we will need to consider other measures for purposes of assessing effectiveness. (4) We employed a greedy algorithm for obtaining "optimal" orderings. This algorithm may not always find the true optimal ordering, and this might allow some heuristic to actually outperform the optimal and generate outliers. However, a true optimal ordering can only be better than the greedy optimal ordering that we utilized; therefore our approach is conservative, and cannot cause us to claim significant differences between optimal and any heuristic where such significance would not exist.

University of Nebraska-Lincoln, Computer Science and Engineering
Technical Report # TR-UNL-CSE-2000-0005
ACM SIGSOFT International Symposium of Software Testing and Analysis, 102-112, August 2000.

5. DISCUSSION

Our results show that there can be statistically significant differences in the rates of fault detection produced by various test case prioritization techniques. To provide a more concrete appreciation for the possible practical consequences of such differences, we illustrate the effects that those differences could have in a specific case.

For this illustration, we have selected, from among our experimental runs, a case (a test suite and version) involving the Space program. We choose this case because it involves a version that contains several (11) faults, and because in this case, optimal and random prioritization created test suites with APFD values close to the mean values exhibited by those techniques on the Space program in our studies. We consider two other prioritization techniques: fn-total and fn-fi-fep-addtl; we select these because in this case they yielded the worst and best prioritization orders, respectively, among the twelve prioritization heuristics. The APFDs for the four techniques, in this run, were: 1) optimal: 99%, 2) fn-fi-fep-addtl: 98%, 3) fn-total: 93%, and 4) random: 84%.[6]

The graph in Figure 4 shows, for these four techniques, the ratio of faults detected as the number of test cases executed increases, and illustrates the differences in fault detection between the runs with differently prioritized test suites. For example, the graph shows that after only 4 of the test cases (2.6% of the test suite) have been run, the optimal ordering has revealed all faults, while the random ordering has revealed only 11.1% of the faults. The fn-total ordering has, in this time, revealed 44.4% of the faults, and the fn-fi-fep-addtl ordering has revealed 77.8%. After six of the test cases (3.9% of the test suite) have been run, both the optimal and fn-fi-fep-addtl orderings have revealed all faults, while fn-total has revealed 44.4% of the faults, and the random ordering has revealed 22.2%. The fn-total and random orderings do not reveal the last faults until 23.7% and 32.9% of the test cases, respectively, have been executed.

Of course, such differences in rate of fault detection are not necessarily of *practical* significance. When the time required to execute all of the test cases in a test suite is short, such differences may be unimportant. When the time required to run all of the test cases in the test suite is sufficiently long, however, these differences may be significant. For example, if the relative fault detection rates exhibited in the above example were mapped onto the testing scenario described in the introduction (with the assumption that test cases have equal costs), in which one of our industrial collaborator's suites requires 7 weeks to execute, then, the differences in rate of detection amount to differences in days, as shown on the scale beneath the graph.

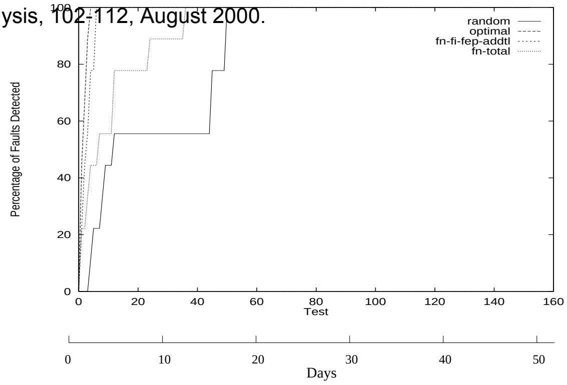


Figure 4: An Example

6. CONCLUSIONS

We have empirically examined the abilities of several test case prioritization techniques to improve the rate of fault detection of test suites. Our studies focus on version-specific test case prioritization, in which test cases are prioritized, and rate of fault detection is measured, relative to specific modified versions of a program.

Our results have several practical consequences. First, our results show that version-specific test case prioritization techniques can improve the rate of fault detection of test suites in regression testing. In fact, all of the techniques we examined, including the simplest ones, can improve the rate of fault detection in comparison to the use of no technique. The fact that these results occur both for function level and statement level techniques is significant because function level techniques are less costly, and involve less intrusive instrumentation, than statement level techniques. However, statement level techniques can produce effectiveness gains, and might thus be preferred if the costs of delays in the detection of faults are sufficiently high. In contrast, our investigation of incorporation of measures of fault proneness into prioritization produced results contrary to our expectations: incorporating these measures did not significantly improve prioritization effectiveness, suggesting that it may be preliminary to attempt to employ them in practice.

Our results also suggest several avenues for future work. First, to address questions of whether these results generalize, further study is necessary. Differences in the performance of the various prioritization techniques we have considered, however, also mandate further study of the factors that underlie the relative effectiveness of various techniques. To address these needs, we are gathering additional programs and gathering and constructing test suites for use in such studies. One desirable outcome of such studies would be techniques for predicting, for particular programs, types of test suites, and classes of modifications, which prioritization techniques would be most effective. We are also investigating alternative prioritization goals and alternative measures of prioritization effectiveness. Finally, because a sizable performance gap remains between prioritization heuristics and optimal prioritization, we are investigating alternative prioritization techniques, including alternative predictors of FEP and fault proneness, and techniques that combine predicted values in different ways.

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

[6] We have deliberately selected an example in which differences between prioritization orders have a significant impact. The example is intended only to provide an appreciation for what differences in APFD values can mean, and for the effect those differences could have, under one testing scenario.

Acknowledgements

This work was supported in part by a grant from Boeing Commercial Airplane Group, and by NSF Faculty Early Career Development Award CCR-9703108, and NSF Award CCR-9707792 to Oregon State University. Siemens Corporate Research shared the Siemens programs. Alberto Pasquini, Phyllis Frankl, and Filip Vokolos shared the Space program and test cases. Roland Untch, Mary Jean Harrold, and Chengyun Chu contributed to earlier stages of this work.

7. REFERENCES

[1] I. S. Association. *Software Engineering Standards*, volume 3 of *Std. 1061: Standard for Software Quality Methodology*. Institute of Electrical and Electronics Engineers, 1999 edition, 1992.

[2] A. Avritzer and E. J. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Trans. on Softw. Eng.*, 21(9):705–716, Sept. 1995.

[3] A. L. Baker, J. M. Bieman, N. Fenton, D. A. Gustafson, A. Melton, and R. Whitty. Philosophy for software measurement. *J. Sys. Softw.*, 12(3):277–281, 1990.

[4] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. In *Proc. of the 3rd Symp. on Softw. Testing, Analysis, and Verification*, pages 210–218, Dec. 1989.

[5] L. C. Briand, J. Wust, S. Ikonomovski, and H. Lounis. Investigating quality factors in object oriented designs: an industrial case study. In *Proc. Int'l. Conf. on Softw. Eng.*, pages 345–354, May 1999.

[6] M. E. Delamaro and J. C. Maldonado. Proteum–A Tool for the Assessment of Test Adequacy for C Programs. In *Proc. of the Conf. on Performability in Computing Sys. (PCS 96)*, pages 79–95, July 1996.

[7] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, Apr. 1978.

[8] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. Technical Report 00-60-03, Oregon State University, Feb. 2000.

[9] S. G. Elbaum and J. C. Munson. A standard for the measurement of C complexity attributes. Technical Report TR-CS-98-02, University of Idaho, Feb. 1998.

[10] S. G. Elbaum and J. C. Munson. Code churn: A measure for estimating the impact of code change. In *Proc. Int'l. Conf. Softw. Maint.*, pages 24–31, Nov. 1998.

[11] S. G. Elbaum and J. C. Munson. Software evolution and the code fault introduction process. *Emp. Softw. Eng. J.*, 4(3):241–262, Sept. 1999.

[12] M. Garey and D. Johnson. *Computers and Intractability*. W.H. Freeman, New York, 1979.

[13] T. Goradia. Dynamic impact analysis: A cost-effective technique to enforce error-propagation. In *ACM Int'l. Symp. on Softw. Testing and Analysis*, pages 171–181, June 1993.

[14] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng.*, SE-3(4):279–290, July 1977.

[15] R. G. Hamlet. Probable correctness theory. *Information Processing Letters*, 25:17–25, Apr. 1987.

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

[17] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. Int'l. Conf. on Softw. Eng.*, pages 191–200, May 1994.

[18] T. M. Khoshgoftaar and J. C. Munson. Predicting software development errors using complexity metrics. *J. on Selected Areas in Comm.*, 8(2):253–261, Feb. 1990.

[19] J. C. Munson. Software measurement: Problems and practice. *Annals of Softw. Eng.*, 1(1):255–285, 1995.

[20] J. C. Munson, S. G. Elbaum, R. M. Karcich, and J. P. Wilcox. Software risk assessment through software measurement and modeling. In *Proc. IEEE Aerospace Conf.*, pages 137–147, Mar. 1998.

[21] A. P. Nikora and J. C. Munson. Software evolution and the fault process. In *Proc. Twenty Third Annual Softw. Eng. Workshop, NASA/Goddard Space Flight Center*, 1998.

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

[23] T. Ostrand and M. Balcer. The category-partition method for specifying and generating functional tests. *Comm. ACM*, 31(6), June 1988.

[24] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Test case prioritization: an empirical study. In *Proc. Int'l. Conf. Softw. Maint.*, pages 179–188, Aug. 1999.

[25] M. Thompson, D. Richardson, and L. Clarke. An information flow model of fault detection. In *ACM Int'l. Symp. on Softw. Testing and Analysis*, pages 182–192, June 1993.

[26] J. Voas. PIE: A dynamic failure-based technique. *IEEE Trans. on Softw. Eng.*, pages 717–727, Aug. 1992.

[27] F. I. Vokolos and P. G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Proc. Int'l. Conf. Softw. Maint.*, pages 44–53, Nov. 1998.

[28] W. Wong, J. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proc. of the Eighth Intl. Symp. on Softw. Rel. Engr.*, pages 230–238, Nov. 1997.