A Controlled Experiment Assessing Test Case Prioritization Techniques via Mutation Faults

Hyunsook Do and Gregg Rothermel
Department of Computer Science and Engineering
University of Nebraska - Lincoln
{dohy, grother}@cse.unl.edu

Abstract

Regression testing is an important part of software maintenance, but it can also be very expensive. To reduce this expense, software testers may prioritize their test cases so that those that are more important are run earlier in the regression testing process. Previous work has shown that prioritization can improve a test suite's rate of fault detection, but the assessment of prioritization techniques has been limited to hand-seeded faults, primarily due to the belief that such faults are more realistic than automatically generated (mutation) faults. A recent empirical study, however, suggests that mutation faults can *be representative of real faults. We have therefore designed and performed a controlled experiment to assess the ability of prioritization techniques to improve the rate of fault detection techniques, measured relative to mutation faults. Our results show that prioritization can be effective relative to the faults considered, and they expose ways in which that effectiveness can vary with characteristics of faults and test suites. We also compare our results to those collected earlier with respect to the relationship between hand-seeded faults and mutation faults, and the implications this has for researchers performing empirical studies of prioritization.*

1 Introduction

As engineers maintain software systems, they periodically regression test them to detect whether new faults have been introduced into previously tested code. Regression testing is an important part of maintenance, but it can also be very expensive, and can account for a large proportion of the software maintenance budget [21]. To assist with regression testing, engineers may prioritize their test cases so that those that are more important are run earlier in the regression testing process.

Test case prioritization techniques schedule test cases for regression testing in an order that attempts to maximize some objective function, such as achieving code coverage quickly, exercising features in order of expected frequency of use, or improving rate of fault detection. Many prioritization techniques have been described in the research literature, and they have been evaluated through various empirical studies [8, 9, 10, 11, 23, 25, 27].

Typically, empirical evaluations of prioritization techniques have focused on assessing a prioritized test suite's rate of detection of *regression faults*: faults created in a system version as a result of code modifications and enhancements. For experimentation, such faults can be obtained in two ways: by locating naturally occurring faults, or by seeding faults. Naturally occurring faults, however, are costly to locate and typically cannot be found in numbers sufficient to support controlled experimentation. In contrast, seeded faults, which are typically produced through hand-seeding or program mutation, can be provided in large numbers, allowing more data to be gathered than otherwise possible.

For these reasons, researchers to date have tended to evaluate regression testing techniques using seeded faults rather than naturally occurring faults. Furthermore, researchers have used hand-seeded faults more frequently than mutation faults, because hand-seeded faults are believed to be more realistic than mutation faults. A recent study by Andrews et al. [1], however, suggests that mutation faults can in fact be representative of real faults. If these results generalize, then we can extend the validity of experimental results on prioritization by using mutation, and the large number of faults that result can yield data sets on which statistically significant conclusions can be obtained, with prospects for assessing causal relationships, and with a relatively low cost compared to hand-seeded faults.

We have therefore performed a controlled experiment to assess prioritization techniques using mutation faults. We examine prioritization effectiveness in terms of rate of fault detection, considering the abilities of several prioritization techniques to improve the rate of fault detection of JUnit test suites on four open-source Java systems, while also varying other factors that affect prioritization effectiveness. Our analyses show that test case prioritization can improve the rate of fault of detection of JUnit test suites, assessed rela-

tive to mutation faults, but the results vary with the numbers of faults and with the test suites' fault detection ability.

In our analysis of results, we also consider our findings in relation to those of Andrews et al., whose study of mutation faults considered only C programs and relative fault detection effectiveness of test suites, without considering the effects on evaluations of client analyses such as prioritization. For the most part, our empirical results are consistent with those of Andrews et al., but they do also suggest that assessments of prioritization techniques could be biased by the use of limited numbers of mutants.

In the next section of this paper, we describe prior work empirically studying prioritization and provide background on program mutation. Section 3 describes the specific mutation operators that we used in this study and our mutant generation process. Section 4 presents our experiment design, results, and analysis. Section 5 discusses our results, and Section 6 presents conclusions and future work.

2 Background and Related Work

2.1 Test Case Prioritization Studies

Early studies of test case prioritization examined the cost-effectiveness of techniques and approaches for estimating technique performance, or compared techniques [11, 23, 25, 27], focusing on C programs. More recent studies have investigated the factors affecting prioritization effectiveness [9, 16, 22], also focusing on C. Collectively, these studies have shown that various techniques can be cost-effective, and suggested tradeoffs among them.

More recently, Saff and Ernst [24] considered test case prioritization for Java in the context of continuous testing, which uses spare CPU resources to continuously run regression tests in the background as a programmer codes. They propose combining the concepts of test frequency and prioritization, and report the results of a study in which prioritized continuous testing reduced wasted development time.

Most recently, Do et al. [8] investigated the effectiveness of prioritization techniques on Java programs tested using JUnit test cases. The results of this study showed that test case prioritization can significantly improve the rate of fault detection of JUnit test suites, but also revealed differences with respect to previous studies that could be related to the language and testing paradigm.

With the exception of one particular C program, a 6000 LOC program from the European Space Agency referred to in the literature as "space", all of the object programs used in this previous empirical work contained only a single type of faults: hand-seeded faults. In contrast, the study we present here assesses prioritization techniques using mutation faults and examines whether the results are consistent with those of the previous study [8] of Java systems tested by JUnit tests, which uses hand-seeded faults.

2.2 Program Mutation

The notion of mutation faults grew out of the notion of mutation testing, a testing technique that evaluates the adequacy of a test suite of a program [5, 6, 13] by inserting simple syntactic code changes into the program, and checking whether the test suite can detect these changes. The effectiveness of mutation testing has been suggested through many empirical studies (e.g., [12, 20]) focusing on procedural languages.

Recently, researchers have begun to investigate mutation testing of object-oriented programs written in Java [4, 17, 18, 19]. While most of this work has focused on implementing object-oriented specific mutant generators, Kim et al. [18] apply mutation faults to some test strategies for object-oriented software and assess them in terms of the effectiveness of object-oriented testing strategies.

Most recently, Andrews et al. [1] investigated the representativeness of mutation faults by comparing the fault detection ability of test suites on hand-seeded, mutation, and real faults, focusing on C systems. Their study finds that mutation faults can in fact be representative of real faults, and thus provide an option for researchers whose their experiments require programs with faults, although more studies are needed to generalize their conclusions.

In this study we further investigate the Andrews et al. findings in the context of test case prioritization using Java programs and JUnit test suites, considering mutation faults and earlier data involving hand seeded faults.

3 Mutation Approach

To conduct our investigation we required a tool for generating program mutants for systems written in Java. The mutation testing techniques described in the previous section use source-code-based mutant generators, but in this study we implemented a mutation tool that generates mutants for Java bytecode. There are benefits associated with this approach. First, it is easier to generate mutants for bytecode than for source code because this does not require parsing source code. Instead, we manipulate Java bytecode using pre-defined libraries contained in BCEL (Byte Code Engineering Library) [3], which provides convenient facilities for analyzing, creating, and manipulating Java class files. Second, because Java is a platform independent language, vendors or programmers might choose to provide just class files for system components, and bytecode mutation lets us handle these files. Third, working at the bytecode level means that we do not need to recompile Java programs after we generate mutants.

3.1 Mutation Operators

To create realistic mutants for Java programs, we surveyed papers that consider mutation testing techniques for

Table 1. Mutation Operators for Java Bytecode

Operators	Descriptions
AOP	Arithmetic Operator Change e.g. + is replaced with -.
LCC	Logical Connector Change e.g. AND is replaced with OR.
ROC	Relational Operator Change e.g. greater than is replaced with less than.
AFC	Access Flag Change e.g. private is replaced with public.
OVD	Overriding Variable Deletion
OVI	Overriding Variable Insertion
OMD	Overriding Method Deletion
AOC	Argument Order Change e.g.A (arg1, arg2) is replaced with A (arg2, arg1).

object-oriented programs [4, 17, 19]. There are many common mutation operators suggested in these papers that handle aspects of object orientation such as inheritance and polymorphism. Among these operators, we selected mutation operators that are applicable to Java bytecode (see Table 1). The first three operators are also typical mutation operators for procedural languages. The other operators are object-oriented specific.

3.2 Mutation Sets for Regression Testing

Because this paper focuses on faults that might be introduced during system evolution, we needed to generate mutants that involve only code modified in moving from one version of a system to a subsequent version. To do this, we built a Java differencing tool that generates a list of names of Java methods that differ from those in a previous version of a program. Our mutant generator can be required to generate mutants using this information. We refer to this mutant generator as a selective mutant generator.

Figure 1 illustrates the selective mutant generation process. The Java differencing tool reads two consecutive versions of a Java source program, P and P', and generates a list of method names (diff-method-name) that are modified from the previous version or newly added to the current version. The selective mutant generator reads diff-method-name and Java class files for P', and generates mutant candidates (Mutant 1, Mutant 2, ..., Mutant k) for P'.

Next, because experimentation with test suites need only consider mutants that can be exposed by those suites, we compared the output of P' run with the tests from our test suites, with and without each mutant present. If the outputs were equivalent for a particular mutant we discarded that mutant. We also discarded mutants that caused "verify" errors during execution, because these represent syntactic errors that would be revealed by simple execution.

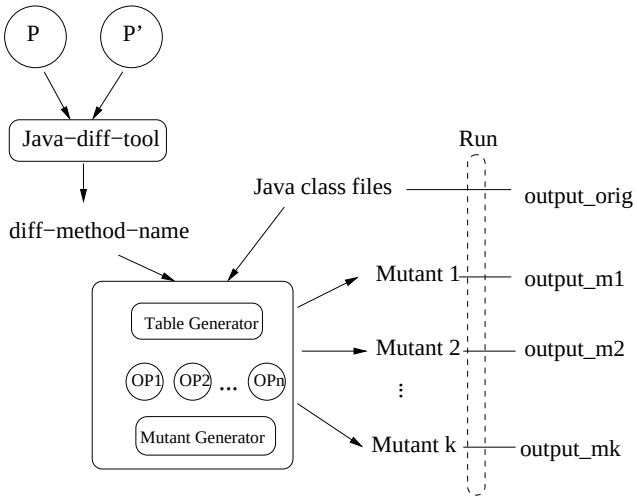

Figure 1. Selective mutant generation process

4 The Experiment

As stated in Section 1, we wished to assess prioritization techniques using mutation faults. In addition to assessing techniques, we also wished to consider whether prioritization results obtained with mutation faults differ from those obtained with hand-seeded faults, and if there is a difference, explore what factors might cause it.

To address our questions we performed a controlled experiment. Because we wished to be able to compare our results to those of our earlier study [8] using hand-seeded faults, our experimental design replicates that of [8].

The following subsections present, for this experiment, our objects of analysis, independent variables, dependent variables and measures, experiment setup and design, threats to validity, and data and analysis.

4.1 Objects of Analysis

We used four Java programs with JUnit test cases as objects of analysis: ant, $xml\text{-}security$, $jmeter$, and $jtopas$. Ant is a Java-based build tool [2]; it is similar to make, but instead of being extended with shell-based commands, it is extended using Java classes. $Jmeter$ is a Java desktop application designed to load test functional behavior and measure performance [14]. $Xml\text{-}security$ implements security standards for XML [28]. $Jtopas$ is a Java library used for parsing text data [15]. All of these programs are publically available as part of an infrastructure supporting experimentation [7].

Table 2 lists, for each of our objects, "No. of versions", "No. of classes", "No. of test cases (test-class level)", "No. of test cases (test-method level)", "No. of faults", "No. of mutants", and "No. of mutant groups". The number of versions is the number of versions of the system that we utilized. The number of classes is the total number of class

Table 2. Experiment Objects and Associated Data

Objects	No. of versions	No. of classes	No. of test cases (test-class level)	No. of test cases (test-method level)	No. of faults	No. of mutants	No. of mutant groups
ant	9	627	150	877	21	2907	187
xml-security	4	143	14	83	6	127	52
jmeter	6	389	28	78	9	295	109
jtopas	4	50	11	128	5	8	7

files in the most recent version of that program. The numbers of test cases at the test-class level and test-method level are the numbers of distinct test cases in the JUnit suites for the programs following two views that will be explained further in Section 4.2.1. The number of faults indicates the total number of hand-seeded faults available for each of the objects. The number of mutants indicates the total number of mutants generated for each of the objects. The number of mutant groups is the total number of sets of mutants that were formed randomly for each of the objects for use in experimentation, and is further explained in Section 4.3.

4.2 Variables and Measures

4.2.1 Independent Variables

Our experiments manipulated two independent variables: prioritization technique and test suite granularity.

Variable 1: Prioritization Technique

We consider seven different test case prioritization techniques, which we classify into three groups to match the earlier study on prioritization for Java programs with hand-seeded faults [8]. Table 3 summarizes these groups and techniques. The first group is the control group, containing three "techniques" that serve as experimental controls. (We use the term "technique" here as a convenience; in actuality, the control group does not involve any practical prioritization heuristics; rather, it involves various orderings against which practical heuristics should be compared.)

Table 3. Test Case Prioritization Techniques.

Label	Mnemonic	Description
T1	untreated	original ordering
T2	random	random ordering
T3	optimal	ordered to optimize rate of fault detection; provides upper bound on the effectiveness of prioritization
T4	block-total	prioritize on coverage of block
T5	block-addtl	prioritize on coverage of block not yet covered
T6	method-total	prioritize on coverage of method
T7	method-addtl	prioritize on coverage of method not yet covered

The other two groups of techniques involve practical heuristics, differring in terms of type of code coverage used. The second group is the block level group, containing two techniques: block-total and block-addtl. By instrumenting a program we can determine, for any test case, the number of basic blocks in that program that are exercised by that test case. We prioritize test cases according to the total number of blocks they cover simply by sorting them in terms of that number. We prioritize test cases in terms of those numbers of additional blocks they cover by greedily selecting the test case that covers the most as-yet-uncovered blocks until all blocks are covered, then repeating this process until all test cases have been used. This second approach, then, incorporates a notion of *feedback* not present in the first approach.

The third group of techniques is the method level group, containing two techniques: method-total and method-addtl. These techniques are the same as corresponding block level techniques except that they rely on coverage measured in terms of methods.

Variable 2: Test Suite Granularity

Test suite granularity measures the number and size of the test cases making up a test suite and can affect the cost of running JUnit test cases, and we want to investigate the relationship between this factor and prioritization technique effectiveness. JUnit test cases are Java classes that contain one or more test methods and that are grouped into test suites, and this provides a natural approach to investigating test suite granularity, by considering JUnit test cases at the test-class level and test-method level. The test-class level treats each JUnit TestCase class as a single test case and the test-method level treats individual test methods within a JUnit TestCase class as test cases. In the normal JUnit framework, the test-class is a minimal unit of test code that can be specified for execution, and provides coarse granularity testing, but by modifying the JUnit framework to handle test-methods individually we can investigate this finer level of granularity.

4.2.2 Dependent Variables and Measures

Rate of Fault Detection

To investigate our research questions we need to measure the benefits of the various prioritization techniques in terms of rate of fault detection. To measure rate of fault detection, we use a metric, APFD (Average Percentage Faults Detected), introduced for this purpose in [11], that mea-

sures the weighted average of the percentage of faults detected over the life of a test suite. APFD values range from 0 to 100; higher numbers imply faster (better) fault detection rates. More formally, let T be a test suite containing n test cases, and let F be a set of m faults revealed by T. Let TF_i be the first test case in ordering T′ of T which reveals fault i. The APFD for test suite T′ is given by the equation:

$$APFD = 1 - \frac{TF_1 + TF_2 + \ldots + TF_m}{nm} + \frac{1}{2n}$$

4.3 Experiment Setup

To perform test case prioritization using mutation faults, we needed to generate mutants and run all mutants on their associated test suites. As described in Section 3, we considered selective mutants, which were created in locations in which code modification occurred relative to the previous version of the program. We compared outputs from program runs in which mutants were enabled (one by one) with outputs from a run of the original program, and selected mutants only if their outputs were different since we were interested only in mutants revealed by test cases.

The numbers of selected mutants derived by this process for our object programs are shown in Table 2. In actual testing scenarios, programs do not typically contain as many faults as these numbers of mutants. Thus, to simulate more realistic scenarios, we randomly selected mutant groups from the pools of mutants created for each program; each mutant group size varied (randomly) between 1 and 5, and no mutant was used in more than one mutant group. Our goal was 30 mutant groups per program version, but some versions of our programs did not have enough mutants to allow formation of this many groups. so our random selection algorithm stopped generating mutant groups for each object when it could not generate any more unique mutant groups, resulting in several cases in which mutant groups numbered less than 30. For example, $jtopas$ has only seven mutant groups across its three versions.

In addition to obtaining mutants, we also needed to collect two types of data to support test case prioritization; namely, coverage information and mutation-fault-matrices. We obtained coverage information by running test cases over instrumented objects using the Galileo system [26] for analysis of Java bytecode in conjunction with a special JUnit adaptor, considering two different instrumentation levels needed by our techniques: all basic blocks and all method entry blocks (blocks prior to the first instruction of the method). This information tracks which test cases exercised which blocks and methods; a previous version's coverage information is used to prioritize the set of test cases for a particular version.

Mutation-fault-matrices list which test cases detect which mutants and are used to measure the rate of fault detection for each prioritization technique.

Since the optimal technique requires information on which test cases expose which mutants in advance to determine an optimal ordering of test cases, it uses mutation-fault-matrices directly when applied.

Each coverage-based prioritization heuristic uses coverage data to prioritize JUnit test suites based on its analysis. APFD scores are then obtained from all reordered test suites. The collected scores are analyzed to determine whether techniques improved the rate of fault detection.

4.4 Threats to Validity

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

Internal Validity

The inferences we have made about the effectiveness of prioritization techniques could have been affected by potential faults in our experiment tools. To control for this threat, we validated our tools on several simple Java programs.

External Validity

Two issues limit the generalization of our results. The first issue is object program representativeness. Our objects are of small and medium size. Complex industrial programs with different characteristics may be subject to different cost-benefit tradeoffs. The second issue involves testing process representativeness. If the testing process we used is not representative of industrial processes, our results might not generalize. Control for these threats can be achieved only through additional studies with wider populations of programs and other testing processes.

Construct Validity

The dependent measure that we have considered, APFD, is not the only possible measure of prioritization effectiveness and has some limitations. For example, APFD assigns no value to subsequent test cases that detect a fault already detected; such inputs may, however, help debuggers isolate the fault, and for that reason might be worth measuring. Also, APFD does not account for the possibility that faults and test cases may have different costs. Future studies will need to consider other measures of effectiveness.

4.5 Data and Analysis

To provide an overview of the collected data we present boxplots in Figure 2. The left side of the figure presents results from test case prioritization applied to the test-class level test cases, and the right side presents results from test case prioritization applied to the test-method level test cases. Each row presents results for each object program.

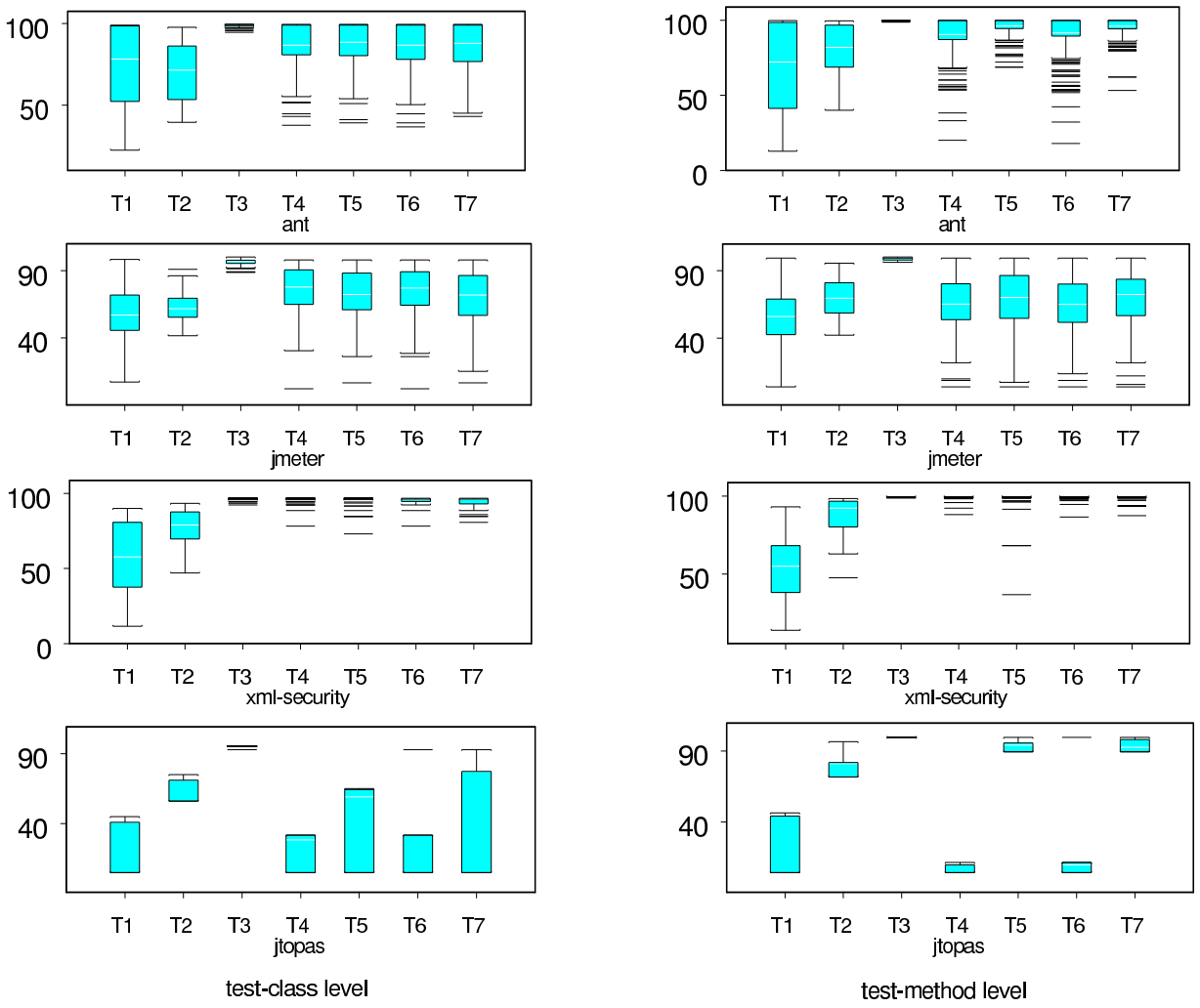

Figure 2. APFD boxplots, all programs, all techniques. The horizontal axes list techniques, and the vertical axes list APFD scores. The left column presents results for test-class level test cases and the right column presents results for test-method level test cases. See Table 3 for a legend of the techniques.

Each plot contains a box for each of the seven prioritization techniques, showing the distribution of APFD scores for that technique across each of the versions of the object program. See Table 3 for a legend of the techniques.

Examining the boxplots for each object program, we observe that the results vary substantially across programs. For example, while the boxplots for $xml\text{-}security$ indicate that the spread of results among non-control techniques is very small for both test suite levels, and prioritization techniques improved the fault detection rate, the boxplots for $jtopas$ show various spreads across techniques and some heuristics are no better than control techniques. For this reason, we analyzed the data for each program separately. For statistical analysis, we used Kruskal-Wallis non-parametric one-way analyses of variance followed by Bonferroni's test for multiple comparisons.[1] For each program, we performed two sets of analyses, considering both test suite levels: untreated vs non-control and random vs non-control. Table 4 presents the results of the Kruskal-Wallis tests, for a significance level of 0.05.

Analysis of results for ant

The boxplots for ant suggest that non-control techniques yielded improvement over (non-optimal) control techniques at both test suite levels. The Kruskal-Wallis test reports that there is a significant difference between techniques for both test suite levels. Thus we performed multiple pair-wise

[1] We used the Kruskal-Wallis test because our data did not meet ANOVA assumptions: our data sets do not have equal variance and some data sets have severe outliers. For multiple comparisons, we used the Bonferroni method for its conservatism and generality.

Table 4. Kruskal-Wallis Test Results, per Program

Program	test suite	control	ch-square	d.f	p-value
ant	test-class	untrtd	56.12	4	< 0.0001
ant	test-meth	untrtd	124.79	4	< 0.0001
ant	test-class	rand	136.69	4	< 0.0001
ant	test-meth	rand	145.04	4	< 0.0001
$jmeter$	test-class	untrtd	71.81	4	< 0.0001
$jmeter$	test-meth	untrtd	37.09	4	< 0.0001
$jmeter$	test-class	rand	55.79	4	< 0.0001
$jmeter$	test-meth	rand	5.38	4	0.2499
$xml\text{-}sec.$	test-class	untrtd	134.68	4	< 0.0001
$xml\text{-}sec.$	test-meth	untrtd	136.18	4	< 0.0001
$xml\text{-}sec.$	test-class	rand	125.1	4	< 0.0001
$xml\text{-}sec.$	test-meth	rand	114.47	4	< 0.0001
$jtopas$	test-class	untrtd	71.86	4	< 0.0001
$jtopas$	test-meth	untrtd	37.09	4	< 0.0001
$jtopas$	test-class	rand	9.52	4	0.0492
$jtopas$	test-meth	rand	16.24	4	< 0.0027

comparisons on the data using the Bonferroni procedure for both test suite levels. The results confirm that non-control techniques improved the rate of fault detection compared to both randomly ordered and untreated test suites.

Regarding the effects of coverage level on prioritization, comparing the boxplots of block-total (T4) to method-total (T6) and block-addtl (T5) to method-addtl (T7), it appears that the level of coverage information utilized (block vs method) had no effect on techniques' rate of fault detection. In contrast, comparing the results of block-total to block-addtl and method-total to method-addtl at both test suite levels, it appears that techniques using feedback do yield improvement over those not using feedback. The Bonferroni analyses confirm these impressions.

Analysis for $jmeter$

The boxplots for $jmeter$ suggest that non-control techniques improved rate of fault detection with respect to randomly ordered and untreated suites at the test-class level, but display fewer differences at the test-method level. The Kruskal-Wallis test reports that there is a significant difference between techniques at both test suite levels with respect to untreated suites, but the analysis for random orderings shows a difference between techniques only at the test-class level. Thus we conducted multiple pair-wise comparisons using the Bonferroni procedure at both test suite levels in the analysis with untreated suites, and at just the test-class level in the analysis with random orderings. The results show that non-control techniques significantly improved the rate of fault detection compared to the (non-optimal) control techniques in all but the case of random at the test-method level.

Regarding the effects of coverage level and feedback, in the boxplots we observe no visible differences between techniques. The Bonferroni analyses confirm that there are no significant differences, at either test suite level, between block-level and method-level coverage, or between techniques that do and do not use feedback.

Analysis for $xml\text{-}security$

The boxplots for $xml\text{-}security$ suggest that non-control techniques were close to optimal with the exception of the presence of outliers. Similar to the results on ant, the Kruskal-Wallis test reports that there are significant differences between techniques at both test suite levels. Thus we conducted multiple pair-wise comparisons using Bonferroni in all cases; the results show that non-control techniques improved the rate of fault detection compared to both random orderings and untreated suites.

Regarding the effects of coverage level and feedback, the results from each technique are very similar, so it is difficult to observe any differences. The Bonferroni analyses revealed no significant differences between block-level and method-level coverage at either test suite level. Techniques using feedback information did perform better than those without feedback at the test-method level, but not at the test-class level.

Analysis for $jtopas$

The boxplots of $jtopas$ are very different from those of the three other programs. It appears that some techniques at the test-method level are better than (non-optimal) control techniques, but other techniques are no better than (non-optimal) control techniques. No prioritization technique produces results better than random orderings at the test-class level. From the Kruskal-Wallis test, for a comparison with random, there is a significant difference between techniques at the test-method level, but just suggestive evidence of difference between techniques at the test-class level (p-value = 0.0492).

The Bonferroni results with random orderings at the test-class level show that there was no significant difference between pairs of techniques. The multiple comparisons with random orders at the test-method level and multiple comparisons with untreated orders at both test suite levels, however, show that techniques using feedback information improved the rate of fault detection compared to random orders and techniques not using feedback information.

Regarding the effects of coverage level and feedback, results were similar to those observed on $jmeter$; the multiple comparisons with each (non-optimal) control technique report that there is no difference between block-level and method-level tests at either test suite level, or between techniques that use feedback and those that do not use feedback at either test suite level.

5 Discussion

Our results show that test case prioritization techniques (assessed using mutation faults) outperformed both un-

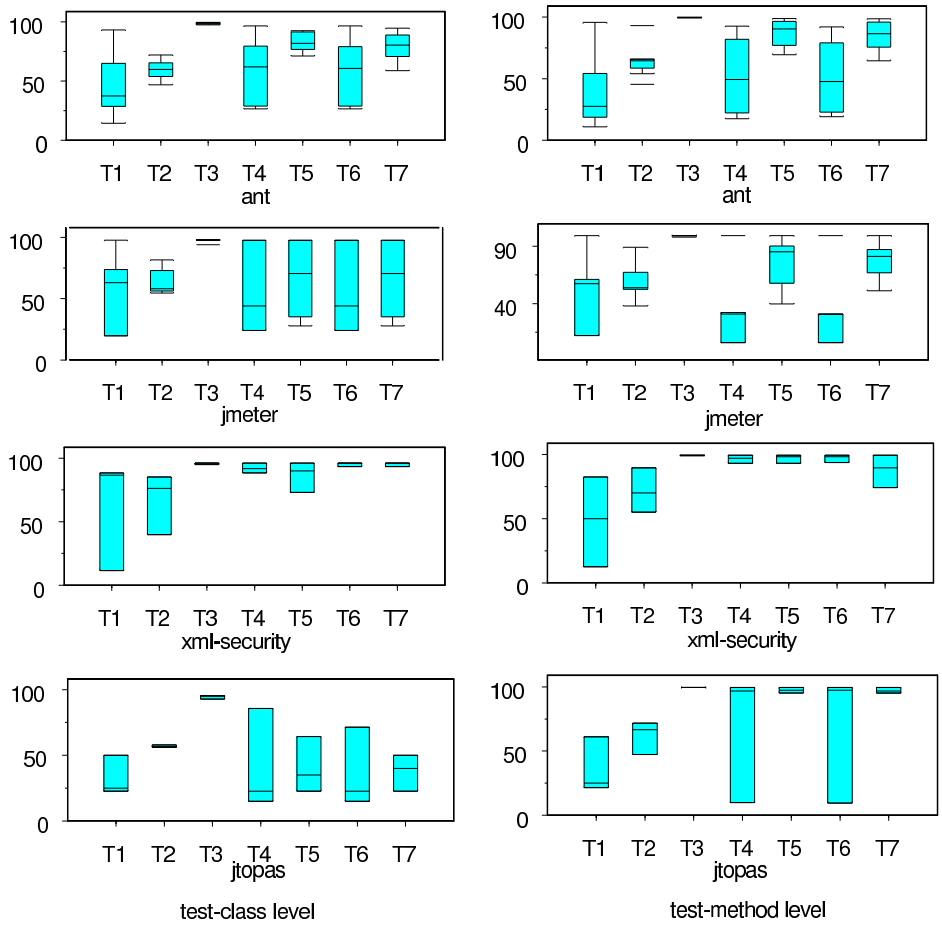

Figure 3. APFD boxplots, all programs, for results with handed-seeded faults (replicated from [8]). The horizontal axes list techniques, and the vertical axes list fault detection rate.

treated and randomly ordered test suites in all but a few cases. Comparing these results with those observed in the earlier study of test case prioritization using hand-seeded faults (see Figure 3) on the same object programs and test suites [8], we observe both similarities and dissimilarities.

First, for all programs, results from this study show less spread of data than those from the study with hand-seeded faults. In particular, the total techniques (T4 and T6) on ant and $jtopas$, and all non-control techniques at the test-class level on $jmeter$, show large differences. This result may be due to the fact that the number of mutants placed in the programs is much larger than the number of seeded faults, which suggests that findings from the study with hand-seeded faults might be biased compared to the study with mutation faults due to larger sampling errors.

Second, results on $jtopas$ show unexpected outcomes unlike those for the other three programs: total coverage techniques are no better than random orderings for both test suite levels, and the data spread is not consistent, showing some similarities with results of the study with hand-seeded faults. We conjecture that this result is due to the small number of mutants that were placed in $jtopas$. In fact, the total number of mutants for $jtopas$, eight, is quite small compared to the numbers of mutants placed in other programs, which varied from 127 to 2907, and is in fact close to the number of hand-seeded faults for the program, five.

Interestingly, the results of this study exhibit trends similar to those seen in studies of prioritization applied to the Siemens and space programs [11], with the exception of results for $jtopas$. Our results contain some outliers, but overall the data distribution patterns for both studies appear similar; with results on $jmeter$ being most similar to results on the Siemens programs. The results for $xml\text{-}security$ are more comparable to those for the space program, showing a small spread of data and high APFD values across all non-control techniques.

From these observations, we infer that studies of prioritization techniques using small numbers of faults may lead to inappropriate assessments. Small data sets, and possibly biased data due to large sampling errors, could significantly affect the legitimacy of findings from experiments.

To investigate how the fault detection abilities of JUnit

test suites on mutation faults and hand-seeded faults differ from each other, and how our findings differ from those of Andrews et al. [1], we measured fault detection rates for our four object programs following Andrews et al.'s experimental procedure. Since the numbers of test cases for our object programs are relatively small compared to those of the Siemens and space programs, we randomly selected 30 test suites of size 10 (without duplication) for each version of each program.

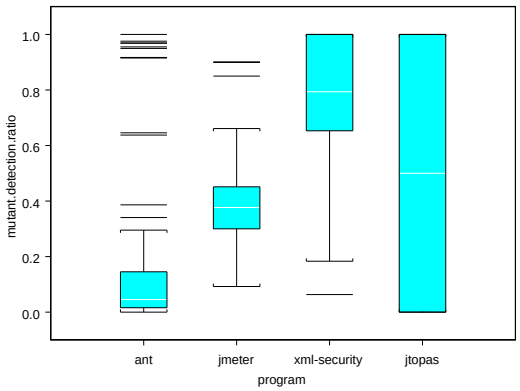

Figure 4. Fault detection ability boxplots for selected small test suites across all program versions. The horizontal axis lists programs, and the vertical axis lists fault detection ratio.

Figure 4 shows the fault detection abilities of these test suites measured on our mutation faults. The vertical axis indicates the fault detection ratio, which is calculated for each test suite S on each program version V by the equation $Dm(S)/Nm(V)$, where $Dm(S)$ is the number of mutants detected by S, and $Nm(V)$ is the total number of mutants in V. Unlike the results of Andrews et al.'s study, our results vary widely across programs. The result for ant shows very low fault detection ability, which means that mutants in ant were difficult to detect, but this might be affected by different factors. For example, test cases for ant do not have strong coverage of the program, and the subsets of these tests that we randomly grouped have relatively little overlapping coverage. We speculate that the latter effect is more a plausible cause of differences since the ant test suite taken as a whole can detect all mutants. In other words, the test suite for ant may have fewer redundant test cases compared to the test suites for the Siemens and space programs.

Results for $xml\text{-}security$ are closer to those of Andrews et al.'s study than those of other programs; note that the distribution of fault detection rates (APFD metric) for $xml\text{-}security$ is similar to that for space. As mentioned in the discussion of results for ant, the test suite for $xml\text{-}security$ might contain many redundant test cases, or each group of test cases might cover more functionality in the program. To further consider this point, we compared the ratio of the number of test cases (at the test-method level) to the number of class files (the size of the program) for ant and $xml\text{-}security$. The last version of ant has 877 test cases and 627 class files (ratio: 877/627 = 1.39), and the last version of $xml\text{-}security$ has 83 test cases and 143 class files (83/143 = 0.58). This means that, proportionally, $xml\text{-}security$ has a smaller number of test cases relative to the test cases for ant, favoring the latter argument as a reason for its higher fault detection ratio.

Result for $jtopas$ have a big spread; this result also is due to the small number of mutants in the program. The first version has only one mutant, so the fault detection ratio for this version can be just two distinct numbers, 0 or 1. The fault detection ratio for $jmeter$ also appears to be low, but it does have a normal distribution with a couple of outliers.

The fault detection ability of hand-seeded faults observed in our earlier study and reconsidered here, overall, is very similar to the result seen on the mutation faults in $jtopas$. We conjecture that this is mainly due to the small numbers of faults in these cases. Even ant, which has the largest number of hand seeded faults in total, displays results similar to those on $jtopas$ with mutation faults, because five out of eight versions of ant contain only one or two faults, and thus the majority of fault detection ratios present 0 or 1 values.

6 Conclusions and Future Work

Studies on the possible usage of mutation faults for controlled experiments with testing techniques have been overlooked prior to the work by Andrews et al. [1]. Whereas Andrews et al. consider the usage of mutation faults on C programs and on the relative fault detection effectiveness of test suites, we consider this issue in the context of a study assessing prioritization techniques using mutation faults, focusing on Java programs.

We have examined prioritization effectiveness in terms of rate of fault detection, considering the abilities of several prioritization techniques to improve the rate of fault detection of JUnit test suites on four open-source Java systems, while also varying other factors that affect prioritization effectiveness. Our analyses show that test case prioritization can improve the rate fault of detection of JUnit test suites, assessed relative to mutation faults, but the results vary with the numbers of mutation faults and with the test suites' fault detection ability.

Our results also revealed similarities and dissimilarities between results from hand-seeded and mutation faults, and in particular, different data spreads between the two were observed. As discussed in Section 5, this difference can be explained in relation to the sizes of the mutation fault sets and hand-seeded fault sets, but more studies and analysis should be done to confirm this fact.

The results of our studies suggest several avenues for fu-

ture work. In particular, we intend to perform additional studies using different types of test suites, such as coverage based and functional test suites, since the fault detection ability of JUnit test suites is different from that of the Siemens and space test suites, which have large pools of coverage-based test suites. We also plan to conduct additional controlled experiments using larger object programs, using different types of mutation faults and testing techniques, to generalize our findings.

Through the results reported in this paper, and our planned future work, we hope to provide useful feedback to testing practitioners wishing to practice prioritization, while also providing alternative choices to researchers who wish to evaluate their testing techniques or testing strategies using various resources that may be available. If our and the Andrews et al. results are generalized through replicated studies, then we can expect significant cost reduction for controlled experiments compared to the cost of experiments with hand-seeded faults.

Acknowledgements

Steve Kachman of the University of Nebraska - Lincoln Statistics Department provided assistance with our statistical analysis. Alex Kinneer and Alexey Malishevsky helped construct parts of the tool infrastructure used in the experimentation. This work was supported in part by NSF under Awards CCR-0080898 and CCR-0347518 to the University of Nebraska - Lincoln.

References

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. Int'l. Conf. Softw. Eng.*, pages 402–411, May 2005.

[2] http://ant.apache.org.

[3] http://jakarta.apache.org/bcel.

[4] J. M. Bieman, S. Ghosh, and R. T. Alexander. A technique for mutation of Java objects. In *Proc. Automated Softw. Eng.*, pages 337–340, Nov. 2001.

[5] T. A. Budd. *Mutation analysis of program test data*. Ph.D. dissertation, Yale University, 1980.

[6] R. A. Demillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. In *IEEE Computer*, pages 34–41, 1978.

[7] H. Do, , S. Elbaum, and G. Rothermel. Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *Proc. Int'l. Symp. Empirical Softw. Eng.*, pages 60–70, Aug. 2004.

[8] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a JUnit testing environment. In *Proc. Int'l. Symp. Softw. Rel. Engr.*, pages 113–124, Nov. 2004.

[9] S. Elbaum, D. Gable, and G. Rothermel. Understanding and measuring the sources of variation in the prioritization of regression test suites. In *Proc. Int'l. Softw. Metrics Symp.*, pages 169–179, Apr. 2001.

[10] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proc. Int'l. Conf. Softw. Eng.*, pages 329–338, May 2001.

[11] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng.*, 28(2):159–182, Feb. 2002.

[12] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. *J. Sys. Softw.*, 38(3):235–253, 1997.

[13] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, 1977.

[14] http://jakarta.apache.org/jmeter.

[15] http://jtopas.sourceforge.net/jtopas.

[16] J. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proc. Int'l. Conf. Softw. Eng.*, pages 119–129, May 2002.

[17] S. Kim, J. A. Clark, and J. A. McDermid. Class mutation: Mutation testing for object-oriented programs. In *Proc. Net.ObjectDays Conf. Object-Oriented Softw. Sys.*, Oct. 2000.

[18] S. Kim, J. A. Clark, and J. A. McDermid. Investigating the effectiveness of object-oriented testing strategies with the mutation method. *J. of Softw. Testing, Verif., and Rel.*, 11(4):207–225, 2001.

[19] Y. Ma, Y. Kwon, and J. Offutt. Inter-class mutation operators for java. In *Proc. Int'l. Symp. Softw. Rel. Engr.*, pages 352–363, Nov. 2002.

[20] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang. An experimental evaluation of data flow and mutation testing. *Softw. Pract. and Exp.*, 26(2):165–176, Feb. 1996.

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

[22] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu. On test suite composition and cost-effective regression testing. *ACM Trans. Softw. Eng. Meth.*, 13(3):227–331, July 2004.

[23] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Trans. Softw. Eng.*, 27(10):929–948, Oct. 2001.

[24] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *Proc. Int'l. Symp. Softw. Rel. Engr.*, pages 281–292, Nov. 2003.

[25] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proc. Int'l. Symp. Softw. Testing Anal.*, pages 97–106, July 2002.

[26] A. Thorat. Galileo: A system for analyzing Java bytecode. MS dissertation, Oregon State University, Jan. 2003.

[27] W. Wong, J. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proc. Int'l. Symp. Softw. Rel. Engr.*, pages 230–238, Nov. 1997.

[28] http://xml.apache.org/security.