On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques

Hyunsook Do, Gregg Rothermel

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

March 27, 2006

Abstract

Regression testing is an important activity in the software lifecycle, but it can also be very expensive. To reduce the cost of regression testing, software testers may prioritize their test cases so that those which are more important, by some measure, are run earlier in the regression testing process. One potential goal of test case prioritization techniques is to increase a test suite's rate of fault detection (how quickly, in a run of its test cases, that test suite can detect faults). 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 primarily to hand-seeded faults, largely 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, and that the use of hand-seeded faults can be problematic for the validity of empirical results focusing on fault detection. We have therefore designed and performed two controlled experiments assessing the ability of prioritization techniques to improve the rate of fault detection of test case prioritization 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. More importantly, a comparison of our results with those collected using hand-seeded faults reveals several implications for researchers performing empirical studies of test case prioritization techniques in particular, and testing techniques in general.

Keywords: regression testing, test case prioritization, program mutation, empirical studies.

1 Introduction

As engineers maintain software systems, they periodically regression test them to detect whether new faults have been introduced into previously tested code, and whether newly added code functions according to specification. Regression testing is an important activity in the software lifecycle, but it can also be very expensive, and can account for a large proportion of the software maintenance budget [31]. 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 (hereafter referred to simply as "prioritization techniques") schedule test cases for regression testing in an order that attempts to maximize some objective function, such as achieving code coverage

quickly, 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 [9, 10, 12, 13, 14, 33, 36, 38, 40].

Typically, empirical evaluations of prioritization techniques have focused on assessing a prioritized test suite's *rate of detection* of *regression faults*. Regression faults are faults created in a system version as a result of code modifications and enhancements, and rate of fault detection is a measure of how quickly a test suite detects faults during the testing process. An improved rate of fault detection can provide earlier feedback on the system under test, enable earlier debugging, and increase the likelihood that, if testing is prematurely halted, those test cases that offer the greatest fault detection ability in the available testing time will have been executed.

When experimenting with prioritization techniques, regression faults can be obtained in two ways: by locating naturally occurring faults or by seeding faults. Naturally occurring faults offer external validity, but they are costly to locate and often 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 larger numbers, allowing more data to be gathered than would otherwise be possible.

For these reasons, researchers to date have tended to evaluate prioritization techniques using seeded faults rather than naturally occurring faults. Furthermore, researchers have typically used hand-seeded faults, because, despite the fact that hand-seeding, too, is costly, hand-seeded faults have been seen as more realistic than mutation faults [17]. A recent study by Andrews et al. [1], however, suggests that mutation faults can in fact be representative of real faults, and that the use of hand-seeded faults can be problematic for the validity of empirical results focusing on fault detection. Their study considered only C programs, and measured only the relative fault detection effectiveness of test suites; it did not consider effects of fault type on evaluations of client testing techniques such as prioritization. If these results generalize, however, then we may be able to improve and extend the validity of experimental results on prioritization by using mutation, and also, to substantially reduce the cost of experimentation, facilitating faster empirical progress.

We have therefore performed two controlled experiments assessing prioritization techniques using mutation faults. In the first experiment, we examine 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. In the second experiment we replicate the first, but we consider a pair of Java programs provided with system, rather than JUnit, test suites.

Our analyses show that test case prioritization can improve the rate of fault detection of test suites, assessed relative to mutation faults, and they expose ways in which that effectiveness can vary with characteristics of faults and test suites, and with classes of prioritization techniques. More important, our empirical results are largely consistent with those of Andrews et al., suggesting that the large number of faults that can be obtained through mutation result in data sets on which statistically significant conclusions can be obtained, with prospects for assessing causal relationships, and with a lower cost compared to that of using hand-seeded faults. The results do also suggest, however, that assessments of prioritization techniques could be biased by the use of overly limited numbers of mutants.

In the next section of this paper, we describe prior work on prioritization and provide background on program mutation. Section 3 examines the current empirical understanding of prioritization as reflected in the literature. Section 4 describes the specific mutation operators that we used in our studies, and our mutant generation process. Sections 5 and 6 present our experiments, including design, results, and analysis. Section 7 discusses our results and considers results across the experiments, and Section 8 presents conclusions and future work.

2 Background and Related Work

2.1 Test Case Prioritization

As mentioned in Section 1, test case prioritization techniques [14, 36, 40] schedule test cases so that those with the highest priority, according to some criterion, are executed earlier in the regression testing process than lower priority test cases. An advantage of prioritization techniques is that unlike many other techniques for assisting regression testing, such as regression test selection [35], they do not discard test cases.

Various prioritization techniques have been proposed [11, 14, 36, 38, 40], but the techniques most prevalent in the literature and in practice involve those that utilize simple code coverage information. In particular, techniques that focus on ordering test cases in terms of code not yet covered by test cases run so far have been shown to be typically most cost-effective, and one such approach has been utilized successfully on extremely large systems at Microsoft [38]. In general, however, the relative cost-effectiveness of these techniques has been shown to vary with several factors. We describe several specific prioritization techniques (those that we study in our experiments) in Section 5.2.1.

Most prioritization techniques proposed to date focus on increasing the rate of fault detection of a prioritized test suite. To measure rate of fault detection a metric called APFD (Average Percentage Faults Detected) has been introduced [14, 36]. This metric measures the weighted average of the percentage of faults detected over the life of a test suite. Section 5.2.2 describes the APFD metric in detail.

Note that to date, most prioritization techniques considered in the literature have focused only on existing test suites, and on obtaining better orderings of the test cases in those suites. A drawback of this focus is that it does not consider the need to add new test cases to test suites following modifications: thus, prioritization of existing test cases should be understood to be only one component of a thorough regression testing process.

2.2 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 [14, 36, 40], focusing on C programs. More recent studies have investigated the factors affecting prioritization effectiveness [10, 21, 34], also focusing on C. Collectively, these studies have shown that various techniques can be cost-effective, and suggested several tradeoffs among them.

More recently, Do et al. [9] 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 appear to be related to the language and testing paradigm.

With the exception of one particular C program, a 6 KLOC program from the European Space Agency referred to in the literature as "space", all of the object programs used in the foregoing empirical work (12 C and 4 Java programs) contained only a single type of faults: hand-seeded faults. In contrast, the studies we present here assess prioritization techniques using mutation faults and examine whether the results are consistent with those of the previous study [9] of Java systems tested by JUnit tests, which used hand-seeded faults.

Beyond these studies, two other studies have considered prioritization relative to actual, non hand-seeded faults [26, 38]. The study in [26] considers prioritization based on the distribution of tests' execution profiles on three large programs, compares results with coverage-based prioritization results, and finds that the two techniques are complementary in terms of fault detection abilities. The study in [38] considers coverage-based prioritization on a large commercial office automation system, and shows how efficiently the prioritization tool works for that system in terms of the time required to prioritize test cases and the speed with which the prioritized test suite can detect faults.

In Section 3, we analyze prior empirical research on prioritization techniques to investigate the relationships that have been seen to exist between the objects used in experiments and prioritization results. We include the studies from [26, 38] in this analysis to help broaden our findings. Results from these two studies, however, are not directly comparable to results obtained in other studies, because the prioritization techniques they use are different, and in the case of the second study, because a different metric is used to measure the effectiveness of prioritization. Thus we consider their results qualitatively.

2.3 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 for a program [5, 7, 16] by inserting simple syntactic code changes into the program, and checking whether the test suite can detect these changes. The potential effectiveness of mutation testing has been suggested through many empirical studies (e.g., [15, 30]) focusing on procedural languages.

Recently, researchers have begun to investigate mutation testing of object-oriented programs written in Java [4, 23, 24, 28]. While most of this work has focused on implementing object-oriented specific mutant generators, Kim et al. [24] apply mutation faults to several testing strategies for object-oriented software, and assess them in terms of the effectiveness of those strategies.

Most recently, as mentioned in Section 1, 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, with results favorable to mutation faults and problematic for hand-seeded faults. Coupled with the fact that mutation faults are much less expensive to produce than hand-seeded faults, mutation faults may provide an attractive alternative for researchers when their experiments require programs with faults. Additional studies are needed, however, to further

generalize this conclusion.

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

3 The State of the Empirical Understanding of Prioritization to Date

Since we are investigating issues related to the types of faults used in experimentation with prioritization techniques, we here analyze prior research that has involved similar experimentation, to provide insights into the relationships that exist between the objects used in experiments and prioritization results.

There have been no prior studies conducted of prioritization using different types of faults over the same programs. Thus, we are not able to directly compare prior empirical results to see whether or not the types of faults utilized could affect results for the same programs; however, we can obtain some general ideas by comparing prioritization results across studies qualitatively. As shown in Section 2, many such studies have been conducted; for this analysis we chose five ([9, 14, 11, 26, 38]) that involve different object programs and types of faults.

Table 1 summarizes characteristics of the object programs used in the five studies we consider. Five of the programs ($javac$, ant, $jmeter$, $xml\text{-}security$, and $jtopas$) are written in Java, and the rest are written in C. Program size varies from 138 LOC to 1.8 million LOC. Six programs (OA, GCC, $Jikes$, $javac$, $space$, and QTB) have real faults, and the others have hand-seeded faults. As a general trend observed in the table, the number of real faults is larger than the number of hand-seeded faults on all programs except $bash$ and some of the $Siemens$ programs. The number of faults for OA was not provided in [38].

Table 1: Object Programs Used in Prioritization Studies

Studies	Program	Program size (LOC)	Versions	Test Cases	Faults	Type of Faults	Prioritization Techniques
Srivastava & Thiagarajan [38]	OA (Office Application)	1.8 million	2	157	-	real	coverage & change-based
Leon & Podgurski [26]	GCC	230K	1	3333	27	real	distribution & coverage-based
	Jikes	94K	1	3149	107	real	
	javac	28K	1	3140	67	real	
Elbaum et al. [14]	Siemens	0.1K-0.5K	1	6-19	7-41	seeded	coverage-based (random, func-total, func-addtl)
	space	6.2K	1	155	35	real	
	grep	7.4k	5	613	11	seeded	
	flex	9K	5	525	18	seeded	
	QTB	300K	6	135	22	real	
Elbaum et al. [11]	bash	65.6K	10	1168	58	seeded	coverage-based (random, func-total, func-addtl)
	gzip	6.5K	6	217	15	seeded	
Do et al. [9]	ant	80.4K	9	877	21	seeded	coverage-based (random, meth-total, meth-addtl)
	jmeter	43.4K	6	78	9	seeded	
	xml-sec.	16.3K	4	83	6	seeded	
	jtopas	5.4K	4	128	5	seeded	

Table 2 shows prioritization results measured using the APFD metric, for all programs except OA, and for four prioritization techniques and one control technique (random ordering) investigated in the papers. The result for OA presents the percentage of faults in the program detected by the first test sequence in the prioritized order. (A test sequence as defined in [38] is a list of test cases that achieves maximal coverage of the program, relative to the coverage achieved by the test suite being prioritized.) In the experiment described in [38], the first sequence contained four test cases, and the entry in Table 2 indicates that those four test cases detected 85% of the faults in the program. For GCC, $Jikes$, and $javac$, a prioritization technique (*comb*) that combines test execution profile distribution and coverage information was applied. For the other programs, two coverage-based prioritization techniques, *total* and *addtl*, which order test cases in terms of their total coverage of program components (functions, methods, or statements), or their coverage of program components not yet covered by test cases already ordered, were applied. As a control technique, a random ordering of test cases was used in all studies other than the one involving OA.

Table 2: Results of Prior Prioritization Studies: Measured Using the APFD Metric, for all Programs Except OA

Technique	OA	GCC	Jikes	javac	Siemens	space	grep	flex	QTB	bash	gzip	ant	jmeter	xml.	jtopas
ccb	85%	-	-	-	-	-	-	-	-	-	-	-	-	-	-
comb.	-	84	74	77	-	-	-	-	-	-	-	-	-	-	-
total	-	-	-	-	86	92	38	66	78	90	50	51	34	97	68
addtl	-	-	-	-	82	94	92	97	67	96	88	84	77	87	97
random	-	80	58	64	68	85	78	88	63	80	75	64	60	71	61

Examining the data in Tables 1 and 2, we observed that the results vary across programs, and thus we further analyzed the data to see what sorts of attributes might have affected these results if any, considering several attributes:

- *Program size*. To investigate whether program size affected the results, we compared results considering three different classes of program size that are applicable to the programs we consider: small (smaller than 10K LOC) – $Siemens$, $space$, $grep$, $flex$, $gzip$, and $jtopas$, medium (larger than 10K LOC and smaller than 100K LOC) – $Jikes$, $javac$, $bash$, ant, $jmeter$, and xml-$security$, and large (larger than 100K LOC) – OA, GCC, and QTB. While large programs are associated with moderate fault detection rates and with prioritization techniques outperforming random ordering, small and medium sized programs do not show any specific trends.

- *Test case source*. The test cases used in the five studies were obtained from one of two different sources: provided with the programs by developers or generated by researchers. Table 3 shows prioritization results grouped by test case source, considering the types of test cases involved (traditional and JUnit) separately. For traditional test suites, prioritization techniques reveal different trends across the two groups: for the provided group, prioritization techniques are always better than random ordering. In particular, $bash$ displays relatively high fault detection rates.

  For the generated group, we can classify programs into two groups relative to results: 1) $Siemens$ and $space$,

Table 3: Prioritization Results Grouped by Test Case Source: Measured Using the APFD Metric, for all Programs Except OA. To Facilitate Interpretation, the Last Row Indicates the Average Number of Faults per Version

	Traditional											JUnit			
	Provided						Generated					Provided			
Tech.	OA	GCC	Jikes	javac	QTB	bash	Siemens	space	grep	flex	gzip	ant	jmeter	xml.	jtopas
ccb	85%	-	-	-	-	-	-	-	-	-	-	-	-	-	-
comb.	-	84	74	77	-	-	-	-	-	-	-	-	-	-	-
total	-	-	-	-	78	90	86	92	38	66	50	51	34	97	68
addtl	-	-	-	-	67	96	82	94	92	97	88	84	77	87	97
random	-	80	58	64	63	80	68	85	78	88	75	64	60	71	61
# faults per ver.	-	27	107	67	3.7	5.8	7-41	35	2.2	3.6	2.5	2.3	1.5	1.5	1.3

2) $grep$, $flex$, and $gzip$. The results on $Siemens$ and $space$ show that prioritization techniques outperform random ordering. Results on the other three programs, however, differ: on these, the total coverage technique does not improve the rate of fault detection, but the additional coverage technique performs well. One possible reason for this difference is that test cases for $Siemens$ and $space$ were created to rigorously achieve complete code coverage of branches and statements. The test cases for the other three programs, in contrast, were created primarily based on the program's functionality, and do not possess strong code coverage.

For JUnit test suites, all of which came with the programs, the results vary depending on program, and it is difficult to see any general trends in these results. In general, however, since JUnit test cases do not focus on code coverage, varying results are not surprising.

- *Number of faults.* On all artifacts equipped with JUnit test suites, as well as on $grep$, $flex$, and $gzip$, the number of faults per version is relatively small compared to on other programs. This, too, may be responsible for the greater variance in prioritization results on the associated programs.

- *Type of faults.* Considering hand-seeded versus real faults, results using real faults show that prioritization techniques always outperform random ordering, but fault detection rates varied across programs. For example, while fault detection rates on $space$ are very high, fault detection rates on QTB, $Jikes$, and $javac$ are relatively low. The study of OA does not use the APFD metric, but from the high fault detection percentage (85%) obtained by the first prioritized test sequence derived for OA and the APFD metric calculation method (see Section 5.2.2), we can infer that the prioritization technique for OA also yields high fault detection rates.

  For programs using hand-seeded faults, results vary across programs. For $Siemens$, $bash$, $xml\text{-}security$, and $jtopas$, prioritization techniques outperform random ordering. For $grep$, $flex$, $gzip$, ant, and $jmeter$, the total coverage technique is not better than random ordering, while the additional coverage technique performs better than random ordering.

- *Other attributes.* We also considered two other attributes that might have affected the results: the type of

Table 4: Mutation Operators for Java Bytecode

Operators	Descriptions
AOP	Arithmetic Operator Change
LCC	Logical Connector Change
ROC	Relational Operator Change
AFC	Access Flag Change
OVD	Overriding Variable Deletion
OVI	Overriding Variable Insertion
OMD	Overriding Method Deletion
AOC	Argument Order Change

language used (C versus Java), and the type of testing being performed (JUnit versus functional), but we can observe no specific trends regarding these attributes.

From the foregoing analysis, we conclude that at least three specific attributes could have affected prioritization results: the type of faults, the number of faults, and the source of test cases. The fact that the type and number of faults could affect prioritization results provides further motivation toward the investigation of the usefulness of mutation faults in empirical investigations of prioritization. Further, this provides motivation for considering evaluations of client testing techniques, and for using different types of faults in relation to the Andrews et al. study. Since the source of test cases could affect prioritization results, some consideration of this factor may also be worthwhile.

4 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 for 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 the parsing of 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.

4.1 Mutation Operators

To create reasonable mutants for Java programs, we surveyed papers that consider mutation testing techniques for object-oriented programs [4, 23, 28]. There are many mutation operators suggested in these papers that handle aspects of object orientation such as inheritance and polymorphism. From among these operators, we selected the following mutation operators that are applicable to Java bytecode (Table 4 summarizes):

Arithmetic OPerator change (AOP). The AOP operator replaces an arithmetic operator with other arithmetic operators. For example, the addition (+) operator is replaced with a subtraction, multiplication, or division operator.

Logical Connector Change (LCC). The LCC operator replaces a logical connector with other logical connectors. For example, the AND connector is replaced with an OR or XOR connector.

Relational Operator Change (ROC). The ROC operator replaces a relational operator with other relational operators. For example, the greater-than-or-equal-to operator is replaced with a less-than-or-equal-to, equal-to, or not-equal-to operator.

Access Flag Change (AFC). The AFC operator replaces an access flag with other flags. For example, this operator changes a private access flag to a public access flag.

Overriding Variable Deletion (OVD). The OVD operator deletes a declaration of overriding variables. This change makes a child class attempt to reference the variable as defined in the parent class.

Overriding Variable Insertion (OVI). The OVI operator causes behavior opposite to that of OVD. The OVI operator inserts variables from a parent class into the child class.

Overriding Method Deletion (OMD). The OMD operator deletes a declaration of an overriding method in a subclass so that the overridden method is referenced.

Argument Order Change (AOC). The AOC operator changes the order of arguments in a method invocation, if there is more than one argument. The change is applied only if arguments have the appropriate type.

The first three of these operators are also typical mutation operators for procedural languages, and the other operators are object-oriented specific.

4.2 Mutation Sets for Regression Testing

Because this paper focuses on regression faults, we needed to generate mutants that involve only code modified in transforming one version of a system to a subsequent version. To do this, we built a tool that generates a list of the names of Java methods, in a version of program P, that differ from those in a previous version of P. Our mutant generator generates mutants using this information. We refer to this mutant generator as a *selective mutant generator*.

Figure 1 illustrates the selective mutant generation process. A differencing tool reads two consecutive versions of a Java source program, P and P', and generates a list of names (diff_method_name) of methods that are modified in P' with respect to P, or newly added to P'. The selective mutant generator reads diff_method_name and Java class files for P', and generates mutants (Mutant_1, Mutant_2, ..., Mutant_k) only in the listed (modified) methods.[1]

[1]The selective mutant generator generates mutants that occur in both the changed code and its neighborhood, where the neighborhood is the enclosing function, and this process matches our hand-seeding process.

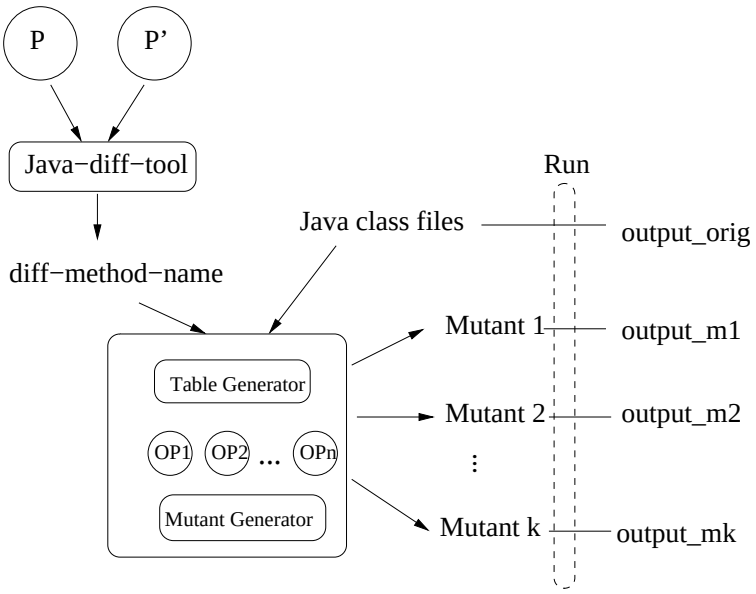


Figure 1: Selective mutant generation process

We then compared outputs from program runs in which these mutants were enabled (one by one) with outputs from a run of the original program, and retained mutants only if their outputs were different. This process is reasonable because we are interested only in mutants that can be revealed by our test cases – since prioritization affects only the rate at which faults that can be revealed by a test suite are detected in a use of that suite. We also discarded mutants that caused verification errors[2] during execution, because these represent errors that would be revealed by any simple execution of the program.

5 Experiment 1

Our primary goal is to replicate prior experiments with prioritization using a new population of faults — mutation faults — in order to consider whether prioritization results obtained with mutation faults differ from those obtained with hand-seeded faults, and if there are differences, explore what factors might be involved in those differences and what implications this may have for empirical studies of prioritization. In doing this, we also gain the opportunity to generalize our empirical knowledge about prioritization techniques, taking into account new study settings.

We begin with a controlled experiment utilizing the same object programs and versions used in an earlier study [9] in which only hand-seeded faults were considered. Our experimental design replicates that of [9]. 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.

[2]As part of the class loading process, a thorough verification of the bytecode in the file being loaded takes place, to ensure that the file holds a valid Java class and does not break any of the rules for class behavior [27].

Table 5: 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
$xml\text{-}security$	4	143	14	83	6	127	52
$jmeter$	6	389	28	78	9	295	109
$jtopas$	4	50	11	128	5	8	7

5.1 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 [18]. $Xml\text{-}security$ implements security standards for XML [41]. $Jtopas$ is a Java library used for parsing text data [19].

These four programs are all provided with hand seeded faults, previously placed in the programs following the procedure described in [9]. Two graduate students performed this fault seeding; they were instructed to insert faults that were as realistic as possible based on their experience with real programs, and that involved code inserted into, or modified in, each of the versions.

All of these programs, along with all artifacts used in the experiment reported here, are publically available as part of an infrastructure supporting experimentation [8].

Table 5 lists, for each of our objects, the following data:

No. of versions. The number of versions of the program that we utilized.

No. of classes. The total number of class files in the latest version of that program.

No. of test cases (test-class level). The numbers of distinct test cases in the JUnit suites for the programs following a test-class level view of testing (this is explained further in Section 5.2.1.)

No. of test cases (test-method level). The numbers of distinct test cases in the JUnit suites for the programs following a test-method level view of testing (this is explained further in Section 5.2.1.)

No. of faults. The total number of hand-seeded faults available (summed across all versions) for each of the objects.

No. of mutants. The total number of mutants generated (summed across all versions) for each of the objects.

No. of mutant groups. The total number of sets of mutants that were formed randomly for each of the objects for use in experimentation (summed across all versions); this is explained further in Section 5.3.

Table 6: Test Case Prioritization Techniques.

Label	Mnemonic	Description
T1	untreated	original ordering
T2	random	random ordering
T3	optimal	ordered to optimize rate of fault detection
T4	block-total	prioritize on coverage of blocks
T5	block-addtl	prioritize on coverage of blocks not yet covered
T6	method-total	prioritize on coverage of methods
T7	method-addtl	prioritize on coverage of methods not yet covered

5.2 Variables and Measures

5.2.1 Independent Variables

The experiment 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; this matches the earlier study on prioritization that we are replicating [9]. The three groups consist of one control group, and two treatment groups that are differentiated by instrumentation levels: block (fine) and method (coarse) levels. Table 6 summarizes these groups and techniques.

The first group is the control group, containing three "orderings" that serve as experimental controls. (We use the term "ordering" here to denote that the control group does not involve any practical prioritization techniques; rather, it involves various test case orderings against which prioritization techniques can be compared.) The untreated ordering is the ordering in which test cases were originally provided with the object. The optimal ordering represents an upper bound on prioritization technique performance, and is obtained by greedily selecting a next test case in terms of its exposure of faults not yet exposed by test cases already ordered. This process is repeated until all test cases are ordered. Ties are broken randomly. (Note that as such, the technique only approximates an optimal ordering.) The random ordering randomly places test cases in order. (To obtain unbiased results for randomly ordered test suites, when obtaining data, we apply twenty independent random orderings for each instance considered and average their results.)

The second group of techniques that we consider 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 (maximal single-entry, single-exit sequences of statements) in that program that are exercised by that test case. The block-total technique prioritizes test cases according to the total number of blocks they cover simply by sorting them in terms of that number. The block-addtl technique prioritizes test cases in terms of the numbers of additional (not-yet-covered) blocks test cases 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 placed in order.

The third group of techniques that we consider is the method level group, containing two techniques: method-total and method-addtl. These techniques are exactly the same as the corresponding block level techniques just described, except that they rely on coverage measured in terms of numbers of methods entered, rather than numbers of blocks covered.

In the remainder of this article, to distinguish the four prioritization techniques in these last two groups from orderings in our control group, we refer to them as "non-control techniques" or "heuristics."

When considering prioritization heuristics such as the four being used here, following prior research [9, 14], techniques can be classified along two orthogonal dimensions. First, techniques are classified in terms of *information type*, where this refers to the type of code coverage information the techniques use. In this study, two information types are considered: method level code coverage information, and block level code coverage information. Second, techniques are classified as incorporating *feedback* when, in the course of prioritizing, they use information about test cases already chosen to select appropriate subsequent test cases. The block-addtl and method-addtl techniques incorporate feedback, whereas the block-total and method-total techniques do not.

Variable 2: Test Suite Granularity

Test suite granularity measures the number and size of the test cases making up a test suite [34] and can affect the cost of running JUnit test cases, and the results of prioritizing them. Following [9], we 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. Note that by construction, a given test-method level test case is smaller than the test-class level test case of what it is a part.

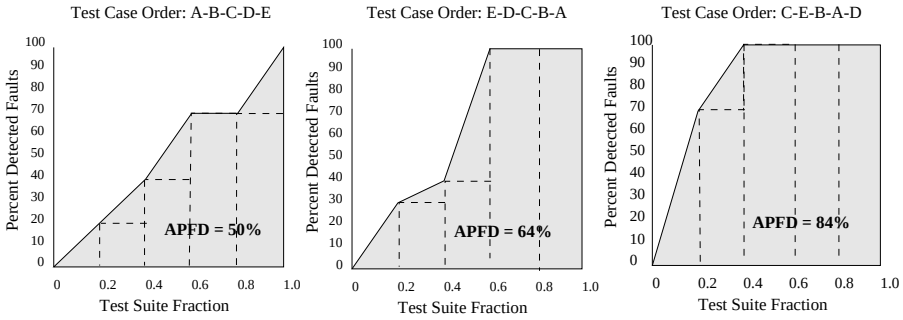
In the normal JUnit framework, the test-class is a minimal unit of test code that can be specified for execution, and provides coarse granularity testing, but by modifying the JUnit framework [20] to be able to treat each test method individually for its execution we can investigate the test-method level of granularity.

5.2.2 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 mentioned in Section 2 called APFD (Average Percentage Faults Detected) [14, 36] that measures 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:

test	fault									
	1	2	3	4	5	6	7	8	9	10
A	x				x					
B						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 2: Examples illustrating the APFD metric.

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

To obtain an intuition for this metric, consider an example program with ten faults and a test suite of five test cases, **A** through **E**, with fault detecting abilities as shown in Figure 2.A. Suppose we place the test cases in order **A–B–C–D–E** to form prioritized test suite $T1$. Figure 2.B shows the percentage of detected faults versus the fraction of $T1$ used. After running test case **A**, two of the ten faults are detected; thus 20% of the faults have been detected after 0.2 of $T1$ has been used. After running test case **B**, two more faults are detected and thus 40% of the faults have been detected after 0.4 of $T1$ has been used. The area under the curve represents the weighted average of the percentage of faults detected over the life of the test suite. This area is the prioritized test suite's average percentage faults detected metric (APFD); the APFD is 50% in this example.

Figure 2.C reflects what happens when the order of test cases is changed to **E–D–C–B–A**, yielding a "faster detecting" suite than $T1$ with APFD 64%. Figure 2.D shows the effects of using a prioritized test suite $T3$ whose test case order is **C–E–B–A–D**. By inspection, it is clear that this order results in the earliest detection of the most faults and illustrates an optimal order, with APFD 84%.

5.3 Experiment Setup

To assess test case prioritization relative to mutation faults, we needed to generate mutants. As described in Section 4, we considered mutants created, selectively, in locations in which code modifications occurred in a program version, relative to the previous version.

The foregoing process created *mutant pools*; one for each version of each object after the first (base) version. The numbers of mutants contained in the mutant pools for our object programs (summed across versions) are shown in Table 5. These mutant pools provide universes of potential program faults. In actual testing scenarios, however, programs do not typically contain faults in numbers as large as the size of these pools. To simulate more realistic testing scenarios, we randomly selected smaller sets of mutants, *mutant groups*, from the mutant pools for each program version. Each mutant group thus selected varied randomly in size between one and five mutants, and no mutant was

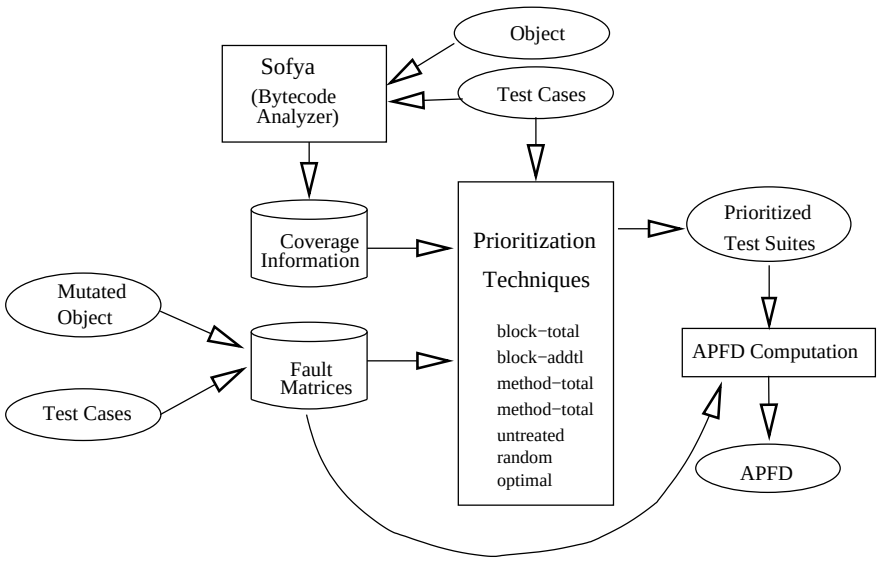


Figure 3: Overview of experiment process.

used in more than one mutant group. We limited the number of mutant groups to 30 per program version, but many versions did not have enough mutants to allow formation of this many groups, so in these cases we stopped generating mutant groups for each object when no additional unique groups could be created. This resulted in several cases in which mutant groups are smaller than 30; for example, $jtopas$ has only seven mutant groups across its three versions.

Given these mutant groups, our experiment then required application of prioritization techniques over each mutant group. The rest of our experiment process is summarized in Figure 3, and proceeded as follows.

First, to support test case prioritization, we needed to collect test coverage information. We obtained coverage information by running test cases over instrumented objects using the Sofya system [25] for analysis of Java bytecode in conjunction with a special JUnit adaptor, considering the 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.

Second, we needed to create fault matrices. Fault-matrices list which test cases detect which mutants and, following approaches used in prior studies [22, 34], were created by running all test cases against each mutant individually.[3]

Third, each prioritization technique was run on each version of each program, with each of that version's test suites. In this step, each coverage-based prioritization heuristic uses coverage data to prioritize test suites based on its analysis. 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. The untreated and random orderings do

[3]As detailed in [22, 34], this approach can under- or overestimate the faults detected by test cases in practice when those faults are simultaneously present in a program, because multiple faults can interact or mask one another. However, it is not computationally feasible to examine all combinations of faults. Fortunately, masking effects have been seen to be limited in prior experiments similar to this one [34].

not require any information to be collected.

Finally, fault matrices are also used in APFD computation to measure the rate of fault detection for each prioritized test suite for each mutant group on each version. The collected scores are analyzed to determine whether techniques improved the rate of fault detection.

5.4 Threats to Validity

Any controlled experiment is subject to threats to validity, and these must be considered in order to assess the meaning and impact of results (see [39] for a general discussion of validity evaluation and a threats classification). In this section we describe the internal, external, and construct threats to the validity of these experiments, and the approaches we used to limit their impact.

External Validity

Two issues affect the generalization of our results. The first issue is the quantity and quality of programs studied. Our objects are of small and medium size. Complex industrial programs with different characteristics may be subject to different cost-benefit tradeoffs. Also, using only four such programs limits the external validity of the results, but the cases in which results are relatively consistent across programs may be places where results generalize. Further, we are able to study a relatively large number of actual, sequential releases of these programs. Nevertheless, replication of these studies on other programs could increase the confidence in our results and help us investigate other factors. Such a replication is provided by the second experiment described in this article.

The second limiting factor is test process representativeness. We have considered only JUnit test suites provided with the objects studied. Complementing these controlled experiments with additional studies using other types of test suites will be necessary. The second experiment described in this article also begins this process.

Internal Validity

To conduct our experiment we required several processes and tools. Some of these processes (e.g., fault seeding) involved programmers and some of the tools were specifically developed for the experiments, all of which could have added variability to our results increasing threats to internal validity. We used several procedures to control these sources of variation. For example, the fault seeding process was performed following a specification so that each programmer operated in a similar way. Also, we validated new tools by testing them on small sample programs and test suites, refining them as we targeted the larger programs.

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.

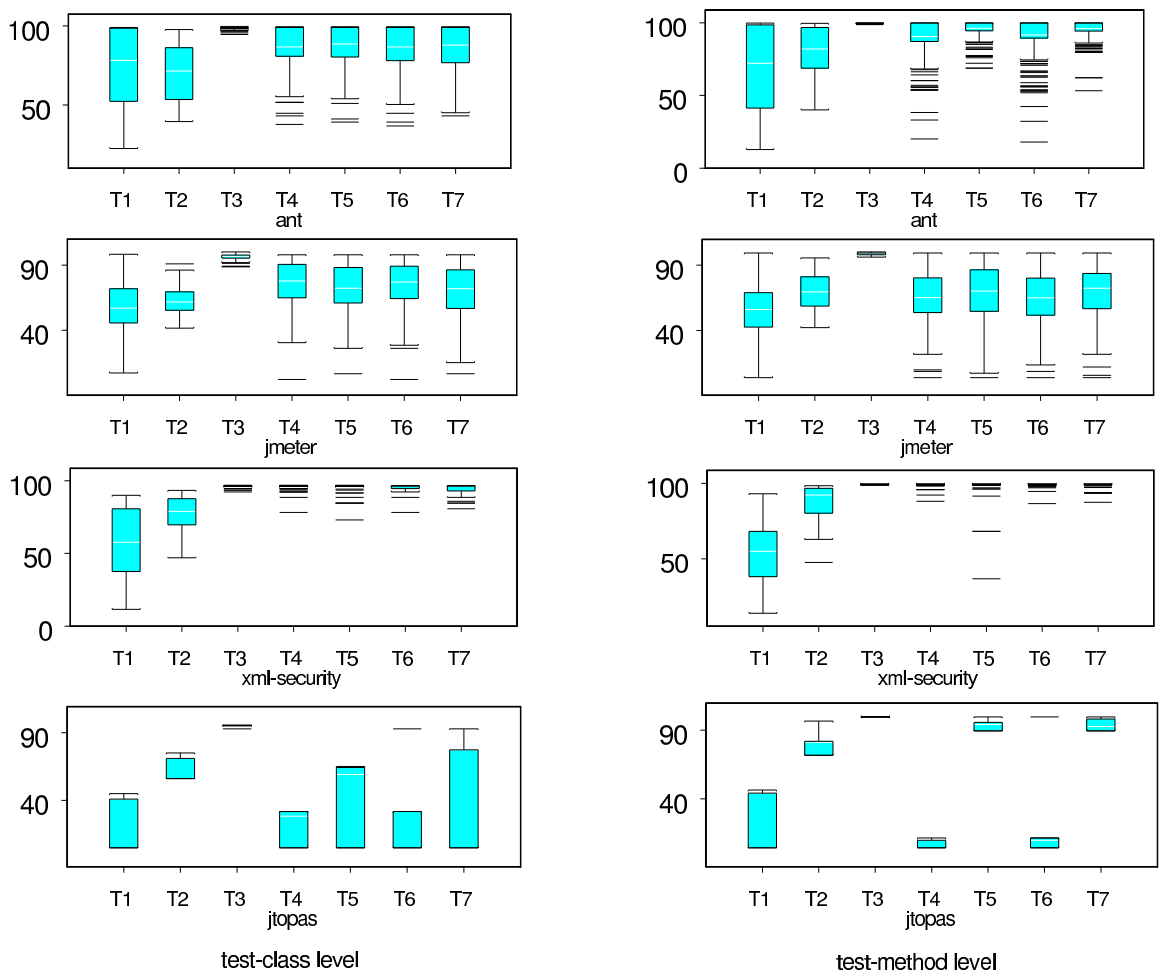


Figure 4: APFD boxplots, all programs, all techniques. The horizontal axes list techniques, and the vertical axes denote APFD scores. The plots on the left present results for test-class level test cases and the plots on the right present results for test-method level test cases. See Table 6 for a legend of the techniques.

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.

Another limiting factor involves our approach to considering test suite granularity. As mentioned in Section 5.2.1, investigating two different test suite granularities (test-class and test-method levels) is a natural approach to use for JUnit test suites, but this is not the only way to consider the sizes of test suites. Future studies will consider alternatives for test suite size as investigated in [34].

5.5 Data and Analysis

To provide an overview of the collected data we present boxplots in Figure 4. The plots on the left side of the figure present results from test case prioritization applied to the test-class level test cases, and the plots on the right side

Table 7: Experiment 1: Kruskal-Wallis Test Results, per Program

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

present results from test case prioritization applied to the test-method level test cases. Each row presents results for one object program. Each plot contains a box for each of the seven prioritization techniques, showing the distribution of APFD scores for that technique across all of the mutant groups used for all of the versions of that object program. See Table 6 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-$security$ indicate that the spread of results among non-control techniques was very small for both test suite levels, and all non-control techniques improved fault detection rate with respect to randomly ordered and untreated test suites, the boxplots for $jtopas$ show various spreads across techniques and some cases in which heuristics were no better than randomly ordered or untreated test suites. For this reason, we analyze the data for each program separately. For each program, following the procedure used in [9], we first consider the data descriptively, and then we statistically analyze the data to (1) compare the heuristics to randomly ordered and untreated test suites, (2) consider the effects of information types on the performance of heuristics, and (3) consider the effects of feedback on the performance of heuristics.

For our statistical analyses, we used the Kruskal-Wallis non-parametric one-way analysis of variance followed (in cases where the Kruskal-Wallis showed significance) by Bonferroni's test for multiple comparisons. (We used the Kruskal-Wallis test because our data did not meet the assumptions necessary for using ANOVA: 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.) We used the Splus statistics package [37] to perform the analyses. For each program, we performed two sets of analyses, considering both test suite levels: untreated versus non-control and random versus non-control. Table 7 presents the results of the Kruskal-Wallis tests, for a significance level of 0.05, and Tables 8 and 9 present the results of the Bonferroni tests at the test-class level and test-method level, respectively. In the two Bonferroni tables, cases in which the differences between techniques compared were statistically significant are marked by "****" (which indicates confidence intervals that do not include zero). Cases in which Bonferroni tests were not performed are marked by "-".

Comparison	*ant*				*jmeter*			
	Estimate	Lower	Upper		Estimate	Lower	Upper	
T1-T4	-14.7	-19.3	-10.0	****	-19.5	-26.5	-12.6	****
T1-T5	-16.1	-20.8	-11.4	****	-14.3	-21.2	-7.3	****
T1-T6	-14.3	-18.9	-9.62	****	-18.3	-25.2	-11.4	****
T1-T7	-15.2	-19.9	-10.6	****	-13.4	-20.3	-6.4	****
T2-T4	-16.6	-20.9	-12.4	****	-13.7	-20.2	-7.2	****
T2-T5	-18.1	-22.4	-13.8	****	-8.4	-15.0	-1.9	****
T2-T6	-16.3	-20.6	-12.0	****	-12.5	-19.0	-5.9	****
T2-T7	-17.2	-21.5	-12.9	****	-7.5	-14.1	-1.0	****
T4-T5	-1.4	-5.7	2.8		5.2	-1.2	11.8	
T4-T6	0.3	-3.9	4.6		1.2	-5.2	7.7	
T4-T7	-0.5	-4.8	3.7		6.1	-0.3	12.7	
T5-T6	1.8	-2.4	6.1		-4.0	-10.5	2.4	
T5-T7	0.8	-3.4	5.1		0.8	-5.6	7.4	
T6-T7	-0.9	-5.2	3.3		4.9	-1.6	11.4	
Comparison	*xml-security*				*jtopas*			
	Estimate	Lower Bound	Upper Bound		Estimate	Lower Bound	Upper Bound	
T1-T4	-35.4	-41.6	-29.1	****	0.6	-43.6	44.8	
T1-T5	-34.7	-41.0	-28.5	****	-16.8	-61.0	27.4	
T1-T6	-35.1	-41.3	-28.8	****	-3.4	-47.6	40.7	
T1-T7	-34.2	-40.5	-28.0	****	-30.1	-74.3	14.1	
T2-T4	-17.7	-21.1	-14.4	****	38.6	-4.0	81.2	
T2-T5	-17.1	-20.4	-13.7	****	21.2	-21.5	63.8	
T2-T6	-17.4	-20.8	-14.1	****	34.5	-8.1	77.1	
T2-T7	-16.6	-19.9	-13.3	****	7.8	-34.8	50.5	
T4-T5	0.6	-2.6	4.0		-17.4	-61.6	26.8	
T4-T6	0.3	-3.0	3.6		-4.0	-48.3	40.1	
T4-T7	1.1	-2.1	4.4		-30.7	-74.9	13.5	
T5-T6	-0.3	-3.6	2.9		13.3	-30.9	57.5	
T5-T7	0.4	-2.8	3.8		-13.3	-57.5	30.9	
T6-T7	0.8	-2.5	4.1		-26.6	-70.8	17.6	

Table 8: Experiment 1: Bonferroni analysis, all programs, test-class level granularity.

5.5.1 Analysis of results for *ant*

The boxplots for *ant* suggest that non-control techniques yielded improvements over random and untreated orderings at both test suite levels. As shown in Table 7, the Kruskal-Wallis test reports that there is a significant difference between techniques for both test suite levels. Thus we performed multiple pair-wise comparisons on the data using the Bonferroni procedure for both test suite levels. The results in Tables 8 and 9 confirm that non-control techniques improved the rate of fault detection compared to both randomly ordered and untreated test suites (as shown in the first eight rows in Tables 8 and 9).

Regarding the effects of information types 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 versus method) had no effect on techniques' rate of fault detection, at both test-method and test-class levels. In contrast, comparing the results of block-total to block-addtl and method-total to method-addtl at the test-method level, it appears that techniques using feedback did yield improvement over those not using feedback. The Bonferroni analyses in Tables 8 and 9 confirm these impressions.

Comparison	*ant*				*jmeter*			
	Estimate	Lower	Upper		Estimate	Lower	Upper	
T1-T4	-21.7	-26.4	-17.1	****	-9.4	-17.0	-2.0	****
T1-T5	-28.6	-33.3	-23.9	****	-12.8	-20.3	-5.3	****
T1-T6	-22.2	-26.8	-17.5	****	-9.7	-17.2	-2.2	****
T1-T7	-28.3	-33.0	-23.7	****	-13.8	-21.3	-6.3	****
T2-T4	-9.7	-13.4	-6.0	****	-	-	-	
T2-T5	-16.6	-20.3	-12.9	****	-	-	-	
T2-T6	-10.2	-13.9	-6.4	****	-	-	-	
T2-T7	-16.3	-20.0	-12.6	****	-	-	-	
T4-T5	-6.8	-10.6	-3.1	****	-3.3	-10.6	3.9	
T4-T6	-0.4	-4.1	3.2		-0.2	-7.4	7.0	
T4-T7	-6.5	-10.3	-2.9	****	-4.3	-11.6	2.9	
T5-T6	6.4	2.7	10.1	****	3.1	-4.1	10.4	
T5-T7	0.2	-3.4	3.9		-0.9	-8.2	6.2	
T6-T7	-6.1	-9.8	-2.4	****	-4.0	-11.3	3.1	
Comparison	*xml-security*				*jtopas*			
	Estimate	Lower Bound	Upper Bound		Estimate	Lower Bound	Upper Bound	
T1-T4	-42.7	-48.9	-36.6	****	9.9e+000	-28.3	48.2	
T1-T5	-40.6	-46.7	-34.5	****	-6.2e+001	-101.0	-24.3	****
T1-T6	-42.9	-49.0	-36.7	****	9.9e+000	-28.3	48.2	
T1-T7	-42.7	-48.8	-36.6	****	-6.2e+001	-100.0	24.0	****
T2-T4	-11.2	-15.1	-7.2	****	5.9e+001	23.2	95.5	****
T2-T5	-9.0	-13.0	-5.0	****	-1.3e+001	-49.3	23.0	
T2-T6	-11.3	-15.2	-7.3	****	5.9e+001	23.2	95.5	****
T2-T7	-11.1	-15.1	-7.1	****	-1.2e+001	-48.9	23.4	
T4-T5	2.1	-1.8	6.0		-7.2e+001	-109.0	-36.3	****
T4-T6	-0.1	-4.0	3.8		2.0e-014	-36.2	36.2	
T4-T7	0.0	-3.9	4.0		-2.2e+001	-108.0	-36.0	****
T5-T6	-2.2	-6.2	1.7		7.2e+001	36.3	109.0	****
T5-T7	-2.0	-6.0	1.8		3.3e+001	35.8	36.5	
T6-T7	0.1	-3.7	4.1		-7.2e+001	-108.0	36.0	****

Table 9: Experiment 2: Bonferroni analysis, all programs, test-method level granularity.

5.5.2 Analysis of results for *jmeter*

The boxplots for *jmeter* suggest that non-control techniques improved rate of fault detection with respect to randomly ordered and untreated test 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 reveals differences 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 random and untreated orderings in all cases other than the one involving random orderings at the test-method level.

Regarding the effects of information types 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.

5.5.3 Analysis of results 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 randomly ordered and untreated test suites.

Regarding the effects of information types and feedback, the results of each technique are very similar, so it is difficult to observe any differences. Similar to results observed on $jmeter$, the Bonferroni analyses revealed no significant differences between block-level and method-level coverage at either test suite level, or between techniques that use and do not use feedback.

5.5.4 Analysis of results for $jtopas$

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

The Bonferroni results with both untreated and random orderings at the test-class level show that there was no significant difference between pairs of techniques. The multiple comparisons at the test-method level, however, show that some non-control techniques improved the rate of fault detection compared to untreated orderings.

Regarding the effects of information types and feedback on prioritization, the multiple comparisons among heuristic techniques report that there is no difference between block-level and method-level tests at either test suite level. Further, techniques using feedback information did outperform those without feedback at the test-method level.

6 Experiment 2

To investigate our research questions further, we replicated Experiment 1 using two additional Java programs with different types of test suites.

6.1 Objects of Analysis

As objects of analysis, we selected two Java programs that are equipped with specification-based test suites constructed using the category-partition method and TSL (Test Specification Language) presented in [32]: $galileo$ and $nanoxml$. *Galileo* is a Java bytecode analyzer, and *nanoxml* is a small XML parser for Java. $Galileo$ was developed by a group of graduate students who created its TSL test suite during its development. $Nanoxml$ was obtained from public

Table 10: Experiment Objects and Associated Data

Objects	No. of versions	No. of classes	No. of test cases	No. of faults	No. of mutants	No. of mutant groups
$galileo$	9	87	1533	35	1568	231
$nanoxml$	6	26	216	33	132	60

domain software and it was not equipped with test cases; so, graduate students created TSL test cases for the program based on its specification, and focusing on its functionality. Both of these programs, along with all artifacts used in the experiment reported here, are publically available [8].

To obtain seeded faults for these programs, we followed the same procedure used originally to seed faults in the Java objects used in Experiment 1, summarized in Section 5.1 and reported in [9].

Table 10 lists, for each of these objects, data similar to that provided for the objects in our first experiment (see Section 5.1); the only exception being that the test suites used for these objects are all system level, and thus, the distinction between test-class and test-method levels does not apply here.

6.2 Variables and Measures

This experiment manipulated just one independent variable, prioritization technique. We consider the same set of prioritization techniques used in Experiment 1 and described in Section 5.2.1. Similarly, as our dependent variable, we use the same metric, APFD, described in Section 5.2.2.

6.3 Experiment Setup

This experiment used the same setup as Experiment 1 (see Section 5.3), but in addition to the steps detailed for that experiment, we also needed to gather prioritization data using our seeded faults since that data was not available from the previous study [9]. We did this following the same procedure given in Section 5.3.

6.4 Threats to Validity

This experiment shares most of the threats to validity detailed for Experiment 1 in Section 5.4, together with additional questions involving the representativeness of the TSL test cases created for the subjects. On the other hand, however, by considering additional objects of study, and a new type of test suites, this experiment helps to generalize those results, reducing threats to external validity.

6.5 Data and Analysis

To provide an overview of the collected data we present boxplots in Figure 5. The left side of the figure presents results from test case prioritization applied to $galileo$, and the right side presents results from test case prioritization applied to $nanoxml$. The upper row presents results for mutation faults, and the lower row presents results for hand-seeded

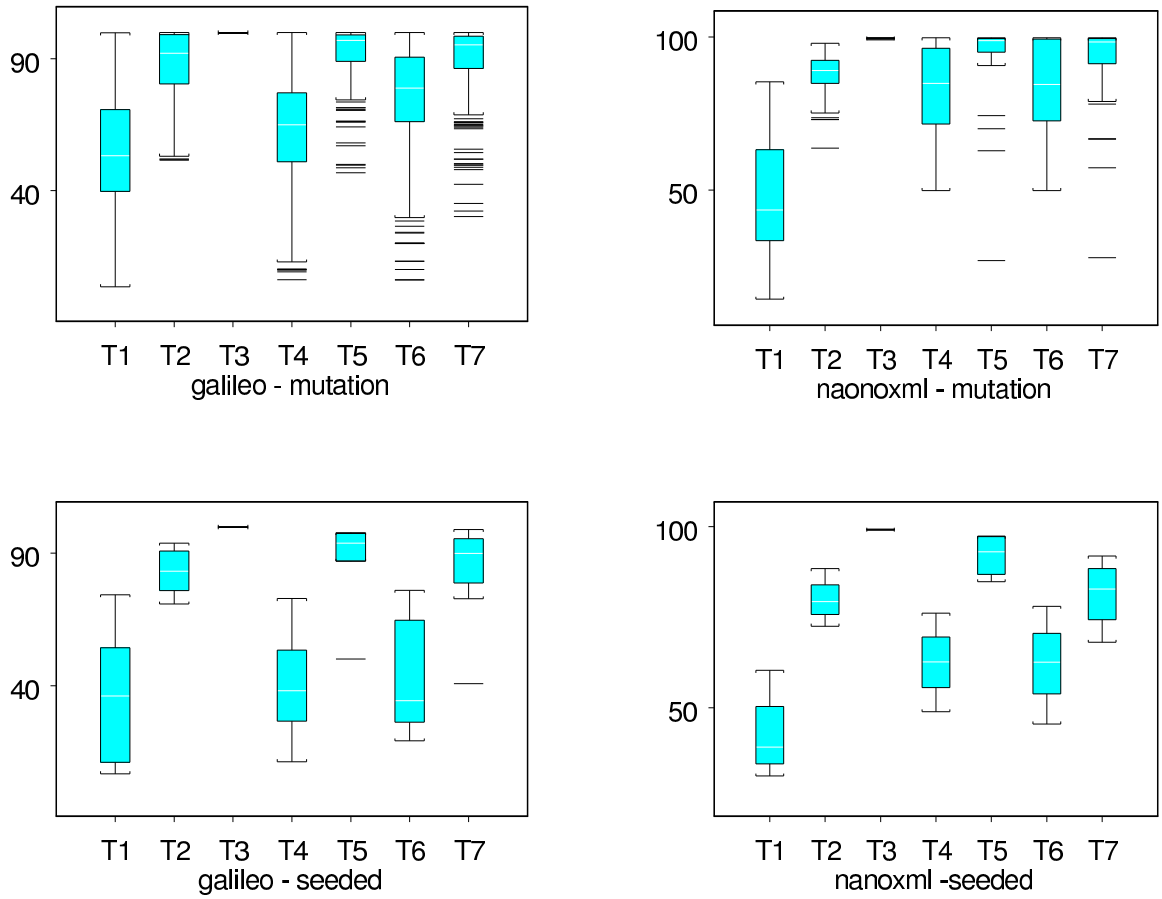


Figure 5: APFD boxplots, all techniques for $galileo$ (left) and $nanoxml$ (right). The horizontal axes list techniques, and the vertical axes list APFD scores. The upper row presents results for mutation faults and the lower row presents results for hand-seeded faults. See Table 6 for a legend of the techniques.

faults. (We postpone discussion of results for hand-seeded faults until Section 7, but we include them in this figure to facilitate comparison at that time.) 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 6 for a legend of the techniques.

Examining the boxplots for each object program, we observe that results on the two programs display several similar trends: all prioritization heuristics outperform untreated test suites, but some heuristics are no better than randomly ordered test suites. Results on $galileo$, however, display more outliers than do results on $nanoxml$, and the variance and skewness in APFD values achieved by corresponding techniques across the two programs differ. For example, APFD values for randomly ordered test suites (T2) show different variance across the programs, and the APFD values from block-total (T4) for $galileo$ appear to form a normal distribution, while they are more skewed for $nanoxml$. For this reason, we analyzed the data for each program separately. For statistical analysis, for reasons

similar to those used in Experiment 1, we used a Kruskal-Wallis non-parametric one-way analysis of variance followed by Bonferroni's test for multiple comparisons. Again, we compared the heuristics to randomly ordered and untreated test suites, in turn, and also considered the effects of information types and feedback on the performance of heuristics. Table 11 presents the results of the Kruskal-Wallis tests, and Table 12 presents the results of the Bonferroni tests.

Table 11: Experiment 2: Kruskal-Wallis Test Results, per Program

Program	control	ch-square	d.f	p-value	Program	control	ch-square	d.f	p-value
$galileo$	untrtd	459.2	4	< 0.0001	$nanoxml$	untrtd	135.3	4	< 0.0001
$galileo$	rand	338.5	4	< 0.0001	$nanoxml$	rand	60.8	4	< 0.0001

Comparison	$galileo$				$nanoxml$			
	Estimate	Lower Bound	Upper Bound		Estimate	Lower Bound	Upper Bound	
T1-T4	-8.6	-13.6	-3.5	****	-32.9	-40.3	-25.5	****
T1-T5	-37.4	-42.4	-32.3	****	-45.6	-53.0	-38.1	****
T1-T6	-20.1	-25.1	-15.0	****	-33.0	-40.5	-25.6	****
T1-T7	-34.5	-39.6	-29.5	****	-43.0	-50.4	-35.6	****
T2-T4	24.7	-7.83	29.0	****	5.0	-1.1	11.2	
T2-T5	-4.1	21.80	0.2		-7.6	-13.8	-1.4	****
T2-T6	13.2	-8.16	17.5	****	4.8	-1.2	11.1	
T2-T7	-1.2	-5.5	3.0		-5.1	-11.3	1.0	
T4-T5	-28.8	-33.1	-24.4	****	-12.7	-18.9	6.5	****
T4-T6	-11.5	-15.8	-7.1	****	-0.1	-6.33	6.0	
T4-T7	-25.9	-30.3	-21.6	****	-10.1	-16.3	-3.9	****
T5-T6	17.3	13.0	21.6	****	12.5	6.35	18.7	****
T5-T7	2.8	-1.4	7.1		2.5	-3.6	8.7	
T6-T7	-14.5	-18.8	-10.1	****	-9.9	-16.2	-3.8	****

Table 12: Experiment 2: Bonferroni analysis, per program.

6.5.1 Analysis of results for $galileo$

The boxplots for $galileo$ suggest that non-control techniques yielded improvement over untreated test suites, and some non-control techniques were slightly better than randomly ordered test suites. The Kruskal-Wallis test (Table 11) reports that there is a significant difference between techniques with respect to untreated and randomly ordered test suites. Thus we performed multiple pair-wise comparisons on the data using the Bonferroni procedure. The results (Table 12) confirm that non-control techniques improved the rate of fault detection compared to untreated test suites. No non-control techniques produced results better than random orderings; however, random orderings outperformed both total techniques overall. (Note, however, that random orderings can often yield worse performance in specific individual runs due to its random nature. The box plots for random orderings show APFD values that are averages of 20 runs for each instance, but individual runs exhibit large variance in APFD values. We discuss this further in Section 7.)

Regarding the effects of information types and their use in 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 an effect on techniques' rate of fault detection for total coverage techniques, but not for additional coverage techniques. Comparing the results of block-total to block-addtl and method-total to method-addtl, it appears that techniques using feedback do yield improvements over those not using feedback. The Bonferroni analyses (Table 12) confirm these impressions.

6.5.2 Analysis of results for $nanoxml$

Similar to results on $galileo$, the boxplots for $nanoxml$ suggest that non-control techniques improved rate of fault detection with respect to untreated test suites. Comparing results from randomly ordered test suites, however, techniques using feedback information appear to improve rate of fault detection, but techniques using total coverage information appear to be worse than randomly ordered test suites. The Kruskal-Wallis test reports that there is a significant difference between techniques with respect to untreated and randomly ordered test suites. Thus we conducted multiple pair-wise comparisons using the Bonferroni procedure. The results show that all non-control techniques significantly improved the rate of fault detection compared to untreated test suites, whereas the only significant difference involving randomly ordered test suites was an improvement associated with block-addtl.

Regarding the effects of information types and feedback and their use in prioritization, the results are the same as those seen on $galileo$, except for one case (block-total (T4) versus method-total (T6)). The Bonferroni analyses confirm these observations.

7 Discussion

To further explore the results of our experiments we consider four topics: (1) a summary of the prioritization results obtained in these experiments and prior studies; (2) an analysis of the differences between mutation and hand-seeded faults with respect to prioritization results; (3) an analysis (replicating the analysis performed by Andrews et al. [1]) of the differences between mutation and hand-seeded faults with respect to fault detection ability; and (4) a discussion of the practical implications of our results.

7.1 Prioritization Results

Results from this study show that non-control prioritization techniques outperformed both untreated and randomly ordered test suites in all but a few cases for JUnit object programs, and outperformed untreated test suites for TSL object programs. The level of coverage information utilized (block vs method) had no effect on techniques' rate of fault detection with one exception on $galileo$ (block-total vs method-total). The effects of feedback information varied across programs: results on ant and $jtopas$ at the test-method level, and on $galileo$ and $nanoxml$, were cases in which techniques using feedback produced improvements over those not using feedback.

Results from our previous prioritization study [9] that used the same set of JUnit programs as those used in Experiment 1 (with hand-seeded faults) also showed that the non-control prioritization techniques we examined outperformed both untreated and randomly ordered test suites, as a whole, at the test-method level. Overall, at the test-class level, non-control prioritization techniques did not improve effectiveness compared to untreated or randomly ordered test suites, but individual comparisons indicated that techniques using additional coverage information did improve the rate of fault detection.

Results from previous studies of C programs [12, 14, 33, 36] showed that non-control prioritization techniques improved the rate of fault detection compared to both random and untreated orderings. Those studies found that techniques using additional coverage information were usually better than other techniques, for both fine and coarse granularity test cases. They also showed that statement-level techniques as a whole were better than function-level techniques.

Interestingly, the results of this study exhibit trends similar to those seen in studies of prioritization applied to the Siemens programs and $space$ [14], with the exception of results for $jtopas$. Our results include 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 $space$, showing a small spread of data and high APFD values across all non-control techniques.

7.2 Mutation versus Hand-Seeded Faults: Prioritization Effects

We next consider the implications, for experimentation on prioritization, of using mutation versus hand-seeded faults.

Our results from Experiment 1 show that non-control test case prioritization techniques (assessed using mutation faults) outperformed both untreated 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 (reproduced from [9] in Figure 6) on the same object programs and test suites, we observe both similarities and dissimilarities.

First, on all programs, results of Experiment 1 often show less spread of data than do results 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 $jmeter$, exhibit large differences. We believe that this result is primarily due to the fact that the number of mutants placed in the programs is much larger than the number of seeded faults, which implies that findings from studies with hand-seeded faults might be biased compared to studies with mutation faults due to larger sampling errors.

Second, results on $jtopas$ differ from results for the other three programs. On $jtopas$, total coverage techniques are no better than random orderings for both test suite levels, and the data spread among techniques is not consistent, showing some similarities with results of the study with hand-seeded faults. We believe 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 much less than the numbers of mutants placed in other programs, which varied from 127 to 2907, and is in fact close to the

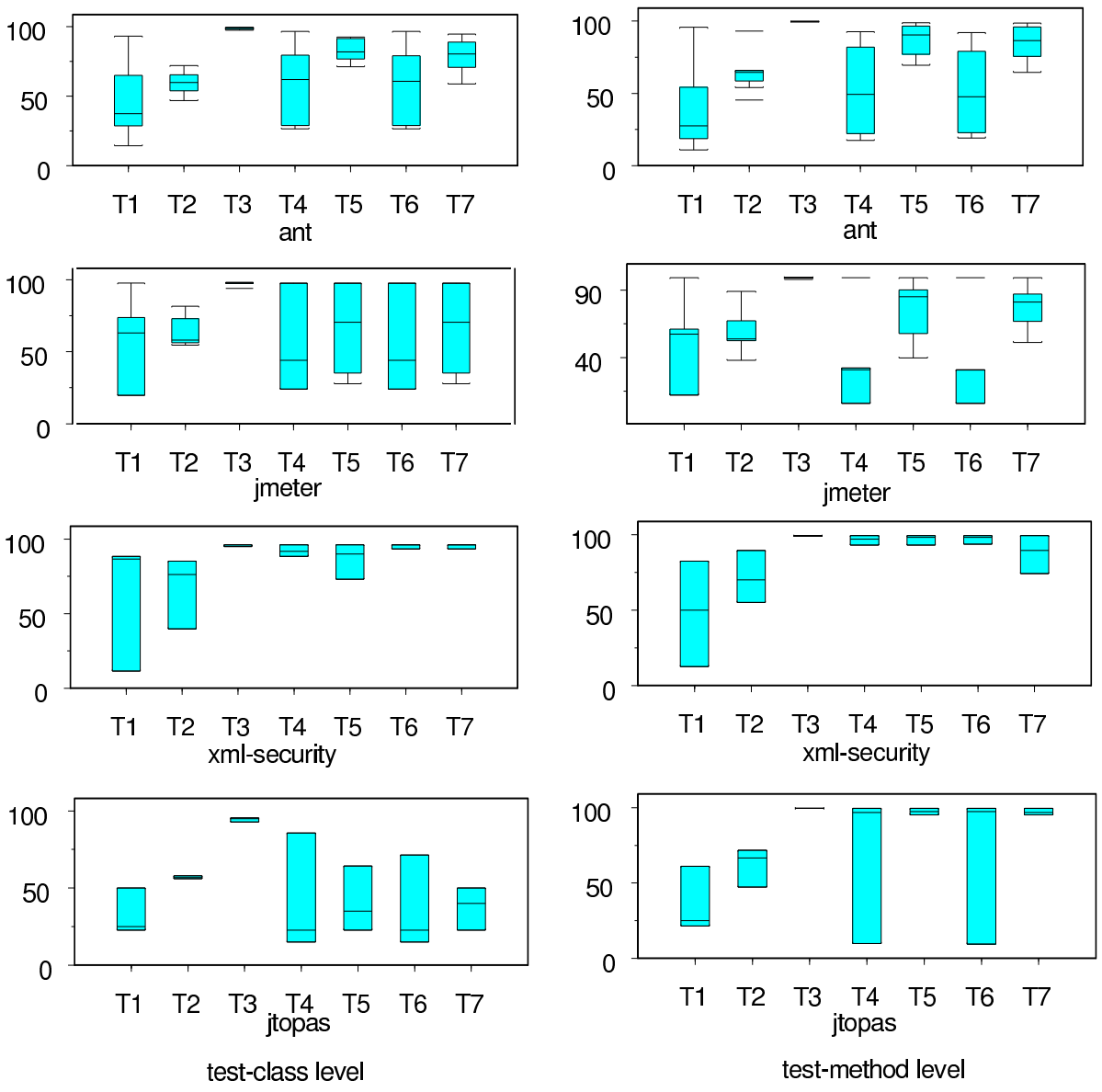


Figure 6: APFD boxplots, all programs, for results with handed-seeded faults (replicated from [9]). The horizontal axes list techniques, and the vertical axes list fault detection rate.

number of hand-seeded faults for the program, five.

Similar to the results of Experiment 1, results of Experiment 2 show that non-control test case prioritization techniques (assessed using mutation faults) outperformed untreated test suites, and some non-control techniques were better than randomly ordered test suites. Comparing these results with those using hand-seeded faults (see Figure 5) on the same object programs and test suites, we also observe both similarities and dissimilarities, and these observations are somewhat different from the observations drawn above.

First, unlike observations drawn from Experiment 1, both results using mutation and hand-seeded faults show similar trends and distribution patterns: all non-control techniques are better than the untreated technique, total coverage techniques are worse than randomly ordered test suites, and the variance between corresponding techniques is not much different. This observation also supports our conjecture regarding the relationship between numbers of faults and prioritization results. $Galileo$ and $nanoxml$ have larger number of hand-seeded faults, 35 and 33, respectively, than the object programs used in the Experiment 1. When we consider the number of hand-seeded faults per version, the difference between two groups of programs persists: while $galileo$ and $nanoxml$ have 3.9 and 5.5 faults per version on average, respectively, JUnit object programs have 2.3, 1.5, 1.5, and 1.4 faults on average per version, respectively.

Overall, results with mutation faults reveal higher fault detection rates and more outliers than those with hand-seeded faults. In particular, the total techniques using mutation faults show more visible differences: for $galileo$, these techniques yield much higher fault detection rates and less spread of data with outliers; for nanoxml, they yield higher fault detection rates, but more spread of data.

The total techniques are worse than randomly ordered test suites for results using both mutation and hand-seeded faults, and this trend is more apparent with hand-seeded faults. One possible reason for this trend is the location of faults in the program and the code coverage ability of test suites that reveal those faults. For example, some faults in $nanoxml$ cause exception handling errors, and thus test cases that reveal those faults tend to have small amounts of code coverage because once a test case reaches the location that causes an exception handling error, then the program execution is terminated with a small portion of code exercised.

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

7.3 Mutation versus Hand-Seeded Faults: Fault Detection Ability

To further understand the results just described, we consider a view of the data similar to that considered by Andrews et al. [1] when comparing mutation to hand-seeded faults, comparing the fault detection abilities of test suites on mutation faults and hand-seeded faults, and noting how our findings differ from those of Andrews et al. [1].

To obtain this view of the data we measured fault detection rates for our six object programs following the experimental procedure used by Andrews et al.. In their study, for each program, 5000 test suites of size 100 were formed by randomly sampling the available test pool.[4] In our case, since the numbers of test cases for our object programs are relatively small compared to those available for the Siemens programs and $space$, we randomly selected between 20 and 100 test suites[5] of size 10 for each version of each program.

Figure 7 shows the fault detection abilities of the test suites created by our sampling process, measured on our mutation and hand-seeded faults. The upper row presents mutation fault detection rates for the four programs used

[4]The authors experimented using various test suite sizes (10, 20, 50, and 100), but these other sizes obtained similar results.
[5]The number of test suites selected varied depending on the number of test cases available in the pools for the programs.

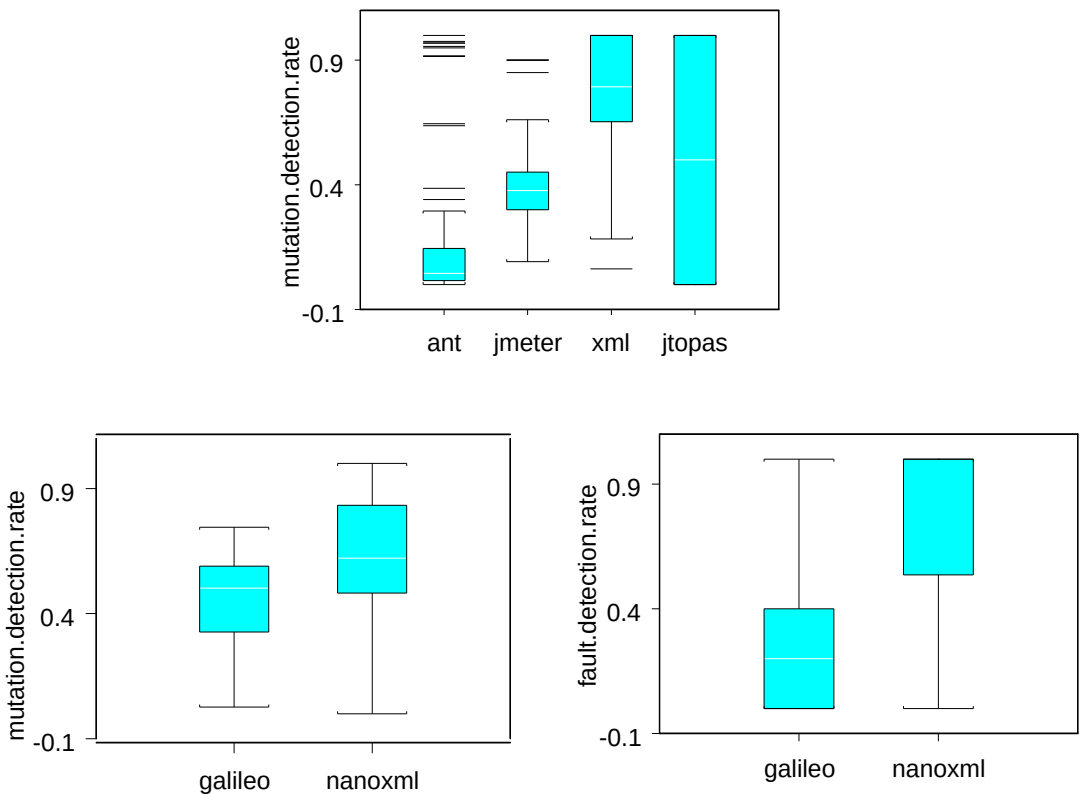


Figure 7: Fault detection ability boxplots for selected small test suites across all program versions. The horizontal axes list techniques, and the vertical axes list fault (or mutant) detection ratios.

in Experiment 1,[6] where JUnit test suites were employed, and the lower row presents results of mutation (left side) and hand-seeded (right side) fault detection rates for the programs used in Experiment 2, where TSL test suites were employed. The vertical axes indicate fault detection ratios, which are 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 the Andrews et al. study, our results vary widely across and between programs with different types of test suites. The result for ant shows relatively low fault detection ability, which means that mutants in ant were relatively difficult to detect, and this might be caused by any of several factors. As two possibilities, test cases for ant do not have strong coverage of the program, and the subsets of these test cases that we randomly grouped have relatively little overlapping coverage. We speculate that the latter effect is a more 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

[6]Since the results of hand-seeded fault detection rates for all four programs are similar to the result for $jtopas$, we omitted them.

coverage-redundant test cases compared to the test suites for the Siemens programs and $space$.

Results for $xml\text{-}security$ are more similar to those of the Andrews et al. study (mean for $space$: 0.75) than those of other programs; the fault detection rates (APFD metric) for $xml\text{-}security$ are similar to those for $space$ (means for func-total and func-addtl are 94 and 96, respectively). 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 than the test cases for ant. To further consider this point, we compared the ratio of the number of test cases for ant (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).[7] This means that, proportionally, $xml\text{-}security$ has a smaller number of test cases relative to program size than ant, favoring the suggestion that each group of test cases might cover more functionality as a reason for its higher fault detection ratio.

Fault detection values for $jtopas$ have a large spread; this result is also 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 similar to the result seen on the mutation faults in $jtopas$. We conjecture that this is primarily 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.

While results among programs with JUnit test suites vary widely, results of mutation detection rates from $galileo$ and $nanoxml$ show more consistent trends. The distributions of detection rates are a bit skewed, but close to a normal distribution, and in particular the fault detection rate distribution for $nanoxml$ is very close to that measured for $space$ in the Andrews et al. study (mean: 0.75, max: 0.82, min: 0.62, 75% :0.77, and 25%: 0.74). The fault detection rate on hand-seeded faults for these two programs also appears different than that of the JUnit object programs. As shown in Figure 7 (lower right), these fault detection rate values are also skewed, but show some variability.

From the two sets of analyses of mutation and hand-seeded fault detection rates, we also observe that the two types of test suites considered are associated with different fault detection rates. The methods used to construct test suites might be a factor in this case: JUnit test suites perform unit tests of Java class files, so the code coverage achieved by these test cases is limited to the class under test. TSL test suites, in contrast, perform functional system-level tests, which cover larger portions of the code than unit tests.

[7] We also measured the ratio using the number of lines of code instead the number of class files, and found results consistent with these: for ant, 877/80.4KLOCs = 10.9 test cases per 1KLOCs; for $xml\text{-}security$, 83/16.3KLOCs = 5 test cases per 1KLOCs.

7.4 Practical Implications

While our results show that there can be statistically significant differences in the rates of fault detection achieved by various test case prioritization techniques applied to Java programs with JUnit and TSL test cases, the improvements seen in rates of fault detection cannot be assumed to be practically significant. Thus, we further consider the effect sizes of differences to see whether the differences we observed through statistical analyses are practically meaningful [29]. Table 13 shows the effect sizes of differences between non-control and control techniques; we calculated these effect sizes only in cases in which the differences between control and non-control techniques showed statistical significance. With the exception of one case (effect size = 0.3 for $galileo$), the effect sizes range from 0.6 to 4.7, which are considered to be large effect sizes [6], so we can say that the differences we observed in this study are indeed practically significant.

Table 13: Experiment Objects and Associated Data

Control	Non-control	Effect Size					
		ant	$jmeter$	$xml\text{-}security$	$jtopas$	$galileo$	$nanoxml$
orig	block-total	1.0	0.5	2.7	-	0.3	2.0
orig	block-addtl	1.4	0.7	2.4	4.7	2.0	2.9
orig	method-total	1.0	0.5	2.7	-	0.9	2.0
orig	method-addtl	1.4	0.7	2.7	4.5	1.7	2.7
rand	block-total	0.6	-	1.4	-	-	-
rand	block-addtl	1.3	-	0.8	-	-	0.8
rand	method-total	0.6	-	1.4	-	-	-
rand	method-addtl	1.2	-	1.4	-	-	-

Even though the foregoing analysis of effect size indicates the practical significance of the differences that we observed in our statistical analyses, a further practical issue to consider is the relationship between the associated costs of prioritization techniques and the benefits of applying them. In practice, prioritization techniques have associated costs, and depending on the testing processes employed and other cost factors, these techniques may not provide savings even though providing higher rates of fault detection.

Our previous study [9] investigated issues involving costs and practical aspects of prioritization results. The study considered models of two different testing processes, batch and incremental testing, and used the resulting models to consider the practical implications for prioritization cost-effectiveness across the different approaches. Further, the study investigated the practical impact of empirical results relative to the four Java programs with JUnit test suites and seeded faults used in Experiment 1, with respect to differences in *delays* values, which represent the cumulative cost of waiting for faults to be exposed while executing a test suite. The study showed that several of the prioritization techniques considered did indeed produce practically significant reductions in delays relative to costs, and thus, that these techniques could be cost-effectively applied in at least certain situations.

Because Experiment 1 utilized the same set of Java programs and JUnit tests for which the foregoing analysis showed practical benefits, the analysis utilized in [9], and its results, are applicable in this case as well. Because our results using mutation faults exhibit better fault detection rates for non-control techniques than those using hand-seeded faults, we expect better practical savings in testing costs relative to these faults in at least the cases considered in Experiment 1.

8 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, however, 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 and TSL test suites on open-source Java systems, while also varying other factors that affect prioritization effectiveness. Our analyses show that non-control test case prioritization can improve the rate of fault detection of both types of 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 reveal similarities and dissimilarities between results using hand-seeded and mutation faults, and in particular, different data spreads between the two were observed. As discussed in Section 7, this difference can be partly 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 further investigate this effect.

More importantly, comparing our results to those collected in earlier studies with hand-seeded faults, our results reveal several implications for researchers performing empirical studies of test case prioritization techniques, and testing techniques in general. In particular, mutation faults may provide a low-cost avenue to obtaining data sets on which statistically significant conclusions can be obtained, with prospects for assessing causal relationships.

For future work we intend to perform various controlled experiments using larger object programs, and using different types of mutation faults and testing techniques, to generalize our findings. We also intend to perform additional controlled experiments that use three different types of faults (real, hand-seeded, and mutation faults) to investigate further the findings by Andrews et al..

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 to 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 *Proceedings of the International Conference on Software Engineering*, 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 *Proceedings of the International Conference on Automated Software Engineering*, pages 337–340, November 2001.

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

[6] R. Coe. What is an effect size?, CEM Centre, Durham University, March 2000.

[7] 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.

[8] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[9] H. Do, G. Rothermel, and A. Kinneer. Prioritizing JUnit test cases: An empirical assessment and cost-benefits analysis. *Empirical Software Engineering: An International Journal*, 11(1):33–70, March 2006.

[10] S. Elbaum, D. Gable, and G. Rothermel. Understanding and measuring the sources of variation in the prioritization of regression test suites. In *Proceedings of the International Software Metrics Symposium*, pages 169–179, April 2001.

[11] S. Elbaum, P. Kallakuri, A. Malishevsky, G. Rothermel, and S. Kanduri. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Journal of Software Testing, Verification and Reliability*, 12(2):65–83, 2003.

[12] S. Elbaum, A. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 102–112, August 2000.

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

[14] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, February 2002.

[15] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. *The Journal of Systems and Software*, 38(3):235–253, 1997.

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

[17] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the International Conference on Software Engineering*, pages 191–200, May 1994.

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

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

[20] http://www.junit.org.

[21] J. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the International Conference on Software Engineering*, pages 119–129, May 2002.

[22] J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test application frequency. In *Proceedings of the International Conference on Software Engineering*, pages 126–135, June 2000.

[23] S. Kim, J. A. Clark, and J. A. McDermid. Class mutation: Mutation testing for object-oriented programs. In *Proceedings of the Net.ObjectDays Conference on Object-Oriented Software Systems*, October 2000.

[24] S. Kim, J. A. Clark, and J. A. McDermid. Investigating the effectiveness of object-oriented testing strategies with the mutation method. *Journal of Software Testing, Verification, and Reliability*, 11(4):207–225, 2001.

[25] A. Kinneer. Assessing the cost-benefits of using type inference algorithms to improve the representation of exceptional control flow in Java. MS thesis, University of Nebraska - Lincoln, August 2005.

[26] D. Leon and A. Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 442–453, November 2003.

[27] C. Lindsey, J. Tolliver, and T. Lindblad. *JavaTech: An Introduction to Scientific and Technical Computing with Java*. Cambridge University Press, 2005.

[28] Y. Ma, Y. Kwon, and J. Offutt. Inter-class mutation operators for java. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 352–363, November 2002.

[29] C. Murphy and B. Myors. *Statistical Power Analysis: A Simple and General Model for Traditional and Modern Hypothesis Tests*. Lawrence Erlbaum Associates, 1998.

[30] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang. An experimental evaluation of data flow and mutation testing. *Software - Practice and Experience*, 26(2):165–176, February 1996.

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

[32] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Comm. ACM*, 31(6):676–686, June 1988.

[33] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia. The impact of test suite granularity on the cost-effectiveness of regression testing. In *Proceedings of the International Conference on Software Engineering*, pages 230–240, May 2002.

[34] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu. On test suite composition and cost-effective regression testing. *ACM Transactions on Software Engineering and Methodology*, 13(3):227–331, July 2004.

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

[36] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.

[37] http://www.insightful.com/products/splus.

[38] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 97–106, July 2002.

[39] C. Wohlin, P. Runeson, M. Host, M. Ohlsoon, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering: An Introduction.* Kluwer Academic Publishers, 2000.

[40] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 230–238, November 1997.

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