

# Understanding the Effects of Changes on the Cost-Effectiveness of Regression Testing Techniques

Sebastian Elbaum,<sup>\*</sup> Praveen Kallakuri,<sup>†</sup> Alexey G. Malishevsky,<sup>‡</sup>  
Gregg Rothermel,<sup>§</sup> Satya Kanduri,<sup>¶</sup>

February 4, 2003

## Abstract

Regression testing is an expensive testing process used to validate modified software. Regression test selection and test case prioritization can reduce the costs of regression testing by selecting a subset of test cases for execution, or scheduling test cases to better meet testing objectives. The cost-effectiveness of these techniques can vary widely, however, and one cause of this variance is the type and magnitude of changes made in producing a new software version. Engineers unaware of the causes and effects of this variance can make poor choices in designing change integration processes, selecting inappropriate regression testing techniques, designing excessively expensive regression test suites, and making unnecessarily costly changes. Engineers aware of causal factors can perform regression testing more cost-effectively. This article reports results of an embedded, multiple case study investigating the modifications made in the evolution of four software systems, and their impact on regression testing techniques. The results of this study expose tradeoffs and constraints that affect the success of techniques, and provide guidelines for designing and managing regression testing processes.

## 1 Introduction

Throughout their lifetimes, software systems undergo numerous changes. Such changes are essential to accommodate new technologies and user needs, but they can also adversely impact the quality and reliability of the software. To address this problem, software test engineers perform regression testing to revalidate software following modifications. Regression testing is important, but it is also expensive. For example, one of the authors of this article works with an industrial collaborator who has, for a software system of only about 20,000 lines of code, a test suite that requires seven CPU weeks to run. A second industrial collaborator, for a much larger system, runs tests continuously, in a rolling test cycle requiring over 30 days. Even much shorter regression test cycles can be expensive, however, when they require human effort to set up for, execute, or validate outputs of tests.

To reduce the costs associated with regression testing, researchers have proposed several techniques. Regression test selection techniques [2, 4, 11, 18, 23, 33] choose a portion of an existing test suite for use in revalidating a software system. Test case prioritization techniques [6, 7, 36, 37] order test cases so that

---

<sup>\*</sup>Department of Computer Science and Engineering, University of Nebraska – Lincoln, [elbaum@cse.unl.edu](mailto:elbaum@cse.unl.edu)

<sup>†</sup>Department of Computer Science and Engineering, University of Nebraska – Lincoln, [pkallaku@cse.unl.edu](mailto:pkallaku@cse.unl.edu)

<sup>‡</sup>Department of Computer Science, Oregon State University, [malishal@cs.orst.edu](mailto:malishal@cs.orst.edu)

<sup>§</sup>Department of Computer Science, Oregon State University, [grother@cs.orst.edu](mailto:grother@cs.orst.edu)

<sup>¶</sup>Department of Computer Science and Engineering, University of Nebraska – Lincoln, [skanduri@cse.unl.edu](mailto:skanduri@cse.unl.edu)

those that are better at achieving testing objectives are run earlier in the regression testing cycle. Empirical studies have shown that these techniques can be cost-effective [3, 4, 6, 8, 12, 34, 35, 36]. These studies have also shown, however, that the cost-effectiveness of prioritization techniques varies with several factors, including the characteristics of the software under test, the attributes of the test cases used in testing, and the modifications made to create the new version of the system.

It is important to understand such factors, and thus, attempts have been made to examine them and their implications for cost-effectiveness. Kim et al. [21] study the impact of regression testing frequency on test selection. Elbaum et al. [5] identify metrics of program structure, test suite composition, and changes that explain some of the variation in prioritization effectiveness. Rothermel et al. [31] study the effects of test case granularity on test selection and prioritization. None of these studies, however, specifically examines how the type and magnitude of changes and their relation to test coverage patterns affect the cost-effectiveness of regression testing techniques.

Engineers unaware of the relationship between change patterns and testing technique cost-effectiveness can make poor choices in defining regression testing processes. These choices may include: (1) designing excessively expensive regression test suites, (2) integrating unnecessarily costly changes into a build, and (3) selecting inappropriate change integration strategies or regression testing techniques. Engineers aware of causal factors could make better choices, and perform regression testing more cost-effectively.

This article presents the results of an embedded multiple case study designed to investigate these issues, by observing the application of several regression testing techniques to several releases of four software systems. The next section of this article reviews the background and previous literature related to this study. Section 3 introduces the goals of the study, and the study’s settings, metrics, threats to validity and results. Section 4 presents the lessons learned and implications for research and practice.

## 2 Background and Related Work

### 2.1 Regression Testing

Let  $P$  be a program, let  $P'$  be a modified version of  $P$ , and let  $T$  be a test suite for  $P$ . Regression testing consists of reusing  $T$  on  $P'$ , and determining where new test cases are needed to effectively test code or functionality added to or changed in producing  $P'$ . Reusing  $T$  and creating new test cases are both important; however, this article focuses on test re-use. In particular, two regression testing methodologies that reuse tests are considered: regression test selection and test case prioritization.

#### 2.1.1 Regression Test Selection

When  $P$  is modified, creating  $P'$ , test engineers may simply reuse all non-obsolete<sup>1</sup> test cases in  $T$  to test  $P'$ ; this is known as the *retest-all* technique [22] and represents typical current practice [27].

The *retest all* technique can be unnecessarily expensive. *Regression test selection* (RTS) techniques [2, 4, 11, 18, 23, 33] ([32] provides a survey) attempt to reduce this cost by using information about  $P$ ,  $P'$ ,

<sup>1</sup>Test cases in  $T$  that no longer apply to  $P'$  are *obsolete*, and must be reformulated or discarded [22].

and  $T$  to select a subset of  $T$  for use in testing  $P'$ . Empirical studies [3, 4, 12, 34, 35] have shown that at least some of these techniques can be cost-effective.

One cost-benefits tradeoff among RTS techniques involves *safety* and *efficiency*. Safe RTS techniques [2, 4, 33] guarantee that, under certain conditions, excluded test cases could not have exposed faults in  $P'$  [32]. Achieving safety, however, may require inclusion of a larger number of test cases than can be run in available testing time. Non-safe RTS techniques, which include techniques that attempt to minimize test cases for coverage [11, 15, 18, 23], sacrifice safety for efficiency, selecting enough test cases to achieve a certain criterion (e.g., coverage adequacy).

### 2.1.2 Test Case Prioritization

Test case prioritization techniques [6, 7, 36, 37] 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. For example, testers might schedule test cases in an order that achieves code coverage at the fastest rate possible, exercises features in order of expected frequency of use, or increases the likelihood that faults will be detected early in testing.

Empirical results [6, 8, 36] suggest that several simple prioritization techniques can significantly improve one testing performance goal: namely, the rate at which test suites detect faults. An improved rate of fault detection during regression testing provides earlier feedback on the system under test and lets software engineers begin locating and correcting faults earlier than might otherwise be possible.

## 2.2 Previous Work

Initially, most studies of regression test selection and test case prioritization focused on the cost-effectiveness of individual techniques, the estimation of a technique's performance, or comparisons of techniques [3, 4, 6, 8, 12, 13, 16, 24, 30, 33, 34, 35, 36, 37]. These studies showed that various techniques could be cost-effective, and suggested tradeoffs among them. However, the studies also revealed wide variances in performance, and attributed these to factors involving the programs under test, the test suites used to test them, and the types of modifications made to the programs.

More recent studies have begun to examine these factors. Rothermel et al. [31] studied the effects of test case granularity on regression testing techniques, varying the composition of test suites and examining the effects on cost-effectiveness of test selection and prioritization. This experiment was performed on two large programs with sequences of versions drawn from the field, containing various modifications. The focus of this experiment, however, was strictly test suite composition: the independent variable manipulated was the number of individual test inputs constituting each test case in the suite. The work did not consider or measure change attributes, and no attempt was made to correlate attributes of change with technique performance.

Two previous studies are more closely related to this one. To investigate the impact of regression testing frequency on test selection, Kim et al. [21] varied the number of changes made in versions of several programs, and measured the impact of this number on test selection and fault detection effectiveness. Elbaum et al. [5]

performed experiments exploring characteristics of program structure, test suite composition, and changes on prioritization, and identified several metrics characterizing these attributes that correlate with prioritization effectiveness.

These studies differ from the work reported in this article in several ways. First, neither of them specifically examined how the type and magnitude of changes, their distribution throughout the program, and their relation to test coverage patterns, affect the cost-effectiveness of regression test selection techniques. Second, although [5] did consider various change characteristics with respect to test case prioritization, it focused on identifying such characteristics, and not on measuring their effects. Moreover, both studies were performed on a set of small programs (an existing suite of seven programs of less than 1K LOC, and one of 6.2K LOC), whose modifications consisted solely of one- or two-line seeded faults, singly or in combination. This constrains the generality of the results, and leaves unresolved questions as to the scalability and applicability of the relationships determined.

### 3 Empirical Study

The overall goal of this study was to determine how the size, distribution, and location of the modifications made to a software system during maintenance affect the effectiveness and efficiency of regression testing techniques, focusing in particular on regression test selection and test case prioritization.

To perform the study, several empirical approaches were considered. A case study rather than a formal experiment was selected, because: (1) four programs were available for study, overcoming the threats to validity raised by previous studies on small systems, (2) it is not possible to control for the evolution of (or modifications made to) the programs studied, or reproduce that evolution at will, and (3) the nature of the questions addressed, which concern how changes impact regression testing methodologies and why technique performance varies under different types of change, are appropriate for a case study [40]. The resulting study employed a multiple-case study design [40], in which each program was studied and analyzed independently (Section 3.4). Then, to render the overall results more robust, repeating trends across the four cases (Section 3.4.5) were examined.

#### 3.1 Study Settings

##### 3.1.1 Methodologies and Techniques

For each of the two regression testing methodologies considered, four techniques were studied, including a control technique, two or three practical heuristics, and where applicable, an optimal technique.<sup>2</sup> This choice of heuristics was motivated by the desire to capture a representative sample of the techniques previously presented in the literature, currently available to practitioners, and applicable to the subject programs studied. The techniques employed are summarized in Table 1, and further details follow.

**Regression Test Selection.** Four regression test selection techniques were selected: *retest-all*, *modified non-core-function*, *modification-focused minimization*, and *coverage-focused minimization*. As described in

---

<sup>2</sup>An optimal technique was applicable only for test case prioritization; the notion of an optimal selected test suite varies with the overall goal of a test selection approach, and such goals themselves vary widely.

<i>Regression Test Selection</i>	<i>Regression Test Prioritization</i>
Retest-all	Random ordering
Modified non-core function	Total function coverage
Modification-focused minimization	Additional function coverage
Coverage-focused minimization	Optimal

Table 1: Methodologies and techniques.

Section 2, the retest-all technique [22] is a safe technique that executes every test case in  $T$  on  $P'$ , and serves as a control technique, representing typical current practice.

The *modified non-core-function* technique adapts a technique defined in [4]; the technique selects test cases that exercise functions in  $P$  that have been deleted or changed in producing  $P'$ , or that exercise functions using variables or structures that have been deleted or changed in producing  $P'$ . The technique tempers its selection, however, by ignoring “core” functions, defined as functions exercised by more than 90% of the test cases in the test suite. This approach trades some degree of safety for additional savings in retesting effort (selecting all test cases through core functions may lead to selecting all of  $T$ ).

The other two approaches utilized are aggressively selective, each seeking (heuristically) “minimal” test suites. The *modification-focused minimization* technique adapts Fischer’s approach [11], which seeks a  $T'$  that is minimal in covering functions in  $P$  identified as changed. The *coverage-focused minimization* technique adapts the test suite reduction technique of Gupta, Harrold, and Soffa [14] to seek a  $T'$  that is minimal in covering all functions in  $P$ .<sup>3</sup> Since the fundamental difference between these techniques involves their consideration (or non-consideration) of modification locations, their choice provides additional perspective on the modification factors that are the focus of this study.

**Test Case Prioritization.** Four test case prioritization techniques were selected: *random ordering*, *total function coverage prioritization*, *additional function coverage prioritization*, and *optimal prioritization*. Random ordering (in this case, by design, equivalent to retest-all) places test cases in  $T$  in random order and acts as a control. Total function coverage prioritization orders test cases in terms of the total number of functions they exercise, resolving ties randomly. Additional function coverage prioritization greedily selects the test case that yields the greatest coverage, then adjusts the coverage data of subsequent test cases based on the functions that were still not covered, and repeats the process. Optimal prioritization uses information on which test cases in  $T$  reveal which faults in  $P'$  to find an optimal ordering for  $T$ ; although not a practical technique (in practice one cannot know which test cases reveal which faults before prioritizing them), this technique provides an upper bound on prioritization benefits. A comprehensive description of these prioritization techniques can be found in [8, 36].

### 3.1.2 Units of Analysis

Several releases of four non-trivial C programs, `bash`, `grep`, `flex`, and `gzip`, were studied. `Bash` is a complete and complex Unix shell, `grep` searches input files for a pattern, `flex` is a lexical analyzer generator, and

<sup>3</sup>In its inception, the technique of [14] is intended to permanently reduce  $T$  for use on all subsequent versions; however, the technique can also be used prior to release of a specific version to selectively retest that version, particularly when change locations are not known, and without permanently discarding test cases; it is in this context that this study considers it.

<i>Program</i>	<i>Version</i>	<i>Functions</i>	<i>Changed Functions</i>	<i>Lines of Code</i>	<i>Regression Faults</i>
bash	2.0	1,494	—	48,292	-
	2.01	1,537	296	49,555	9
	2.01.1	1,538	44	49,666	7
	2.02	1,678	296	58,090	7
	2.02.1	1,678	12	58,103	3
	2.03	1,703	188	59,010	9
	2.04	1,890	339	63,802	5
	2.05-b1	1,942	447	65,477	6
	2.05-b2	1,949	40	65,591	7
	2.05	1,950	27	65,632	5
flex	2.4.3	139	-	8,254	-
	2.4.7	149	40	8,695	5
	2.5.1	165	93	10,297	4
	2.5.2	165	26	10,326	7
	2.5.3	166	12	10,416	1
grep	2.0	130	-	8,164	-
	2.2	172	73	11,946	4
	2.3	175	37	12,682	3
	2.4	177	29	12,781	3
	2.4.1	182	109	13,282	1
gzip	1.0.7	86	-	4,744	-
	1.1.2	86	37	5,228	5
	1.2.2	103	26	5,811	3
	1.2.3	102	14	5,727	2
	1.2.4	102	32	5,810	2
	1.3	108	52	6,582	3

Table 2: Experiment Subjects.

`gzip` is a compression and decompression utility. Table 2 lists, for each version of each program, the numbers of functions, changed functions (functions modified or added to the version since the preceding version, or deleted from the preceding version), and non-comment, non-blank lines of code (referred to in the rest of this article as “executable lines of code”). (The rightmost column is described later.)

The study required that test suites be available for the subjects. For `grep`, `flex`, and `gzip` no such suites were available. To construct test cases representative of those that might be created in practice for these programs, a process defined initially by Hutchins et al. [20] for use in studying test adequacy criteria was adapted. The documentation of the programs, and the parameters and special effects determined to be associated with each program, were considered informal specifications. These informal specifications, together with the category partition method and an implementation of the TSL tool [1, 28], were used to construct a suite of test cases that exercise each parameter, special effect and erroneous condition affecting program behavior. Those test suites were then augmented with additional test cases to increase code coverage (measured at the statement level). These suites were created for the base versions of the programs; they served as regression suites for subsequent versions.

`Bash` was somewhat different from the other three subject programs, in that each version of `Bash` had been released with a test suite, composed of test cases from previous versions and new test cases designed to validate added functionality. Some of these test suites, however, contained test cases that were obsolete for

<i>Program</i>	<i>Version</i>	<i>Number of Test Cases</i>	<i>Functional Coverage</i>
bash	2.0	1168	64
flex	2.4.3	525	86
grep	2.0	613	70
gzip	1.07	217	89

Table 3: Test Suites.

later versions of the system. Thus, the test cases from release 2.0 of the system were selected, because most of these test cases worked on all releases. This suite was then updated to test additional features (considering the reference documentation for **Bash** [29] as an informal specification) and increase coverage of functions.

The characteristics of the test suites created for the four programs are presented in Table 3, and include the number of test cases in the suite, and the percentage of program functions covered by the test cases on the baseline version (oldest version available for each program).

### 3.1.3 Faults

One goal of the study was to evaluate the performance of regression testing techniques with respect to detection of regression faults, that is, faults created in a program version as a result of the modifications that produced that version. Such faults were not available with the programs studied; thus, to obtain them, a procedure similar to one defined and employed in several previous studies of testing techniques [20, 38, 39] was used, as follows. The two labs involved in the study recruited graduate and undergraduate students in computer science with at least two years of C programming experience. These students were instructed to insert faults that were as realistic as possible based on their experience, and that involved code deleted from, inserted into, or modified between the versions. To further direct their efforts, the fault seeders were given the following list of types of faults to consider: <sup>4</sup>

- Faults associated with variables, such as definitions of variables, redefinitions of variables, deletions of variables or changes in values of variables in assignment statements;
- Faults associated with control flow, such as addition of new blocks of code, deletions of paths, redefinitions of execution conditions, removal of blocks, changes in order of execution, new calls to external functions, removal of calls to external functions, addition of functions or deletions of functions;
- Faults associated with memory allocation, such as not freeing allocated memory, failing to initialize memory or creating erroneous pointers.

Ten potential faults were seeded in each version of each program; then these faults were activated individually, and the test suites for the programs were executed to determine which faults could be revealed by which test cases. Any potential faults that were not detected by any test cases were excluded: such faults are meaningless to the measures used in this study and have no bearing on the results. Also excluded were any faults that were detected by more than 25% of the test cases, under the assumption that such easily detected

<sup>4</sup>These directives were adapted from the fault classification used by Nikora et al. [26].

faults would be detected by engineers during their unit testing of modifications. This process yielded a total of 101 regression faults, which are reported in the rightmost column of Table 2.

### 3.1.4 Additional Instrumentation

To perform the experiments additional instrumentation was needed. Test coverage information was provided by the Aristotle program analysis system [17] and by the Clic instrumentor and monitor [9]. The selected test case prioritization and regression test selection techniques were implemented. Unix utilities and direct inspection were used to determine modified functions and compute the required change metrics.

## 3.2 Variables

### 3.2.1 Independent variables

For each program considered there were two independent variables: the chosen regression testing technique and the program version with its particular changes. The techniques employed for regression test selection and prioritization were described in Section 3.1.1. The second independent variable, change, was quantified along three dimensions: size, distribution, and coverage, as follows:

- *P-CH-L: percentage of changed lines of code.* P-CH-L measures the fraction of the total executable lines of code that were added, changed, or removed across all functions between a version and its predecessor, and indicates the size of the change made to that version.
- *P-CH-F: percentage of changed functions.* P-CH-F measures the fraction of the total number of functions that contained at least one line changed between a version and its predecessor. To reduce a possible confounding influence caused by test suite coverage, only the changed functions that are present in both versions and that are covered by the test suite were counted. A higher value for P-CH-F indicates higher change distribution, and possibly higher change size.
- *A-CH-LOC-F: average number of lines of code changed per function.* A-CH-LOC-F measures the average amount of change per function in a given version, obtained by dividing the number of changed lines of code by the number of changed functions. High values of this metric indicate larger changes per function.
- *P-CH-Files: percentage of changed files.* P-CH-Files measures the fraction of the total number of files that changed in a given version. A file is considered to have been changed if at least one constituent function changed from the previous version. This value is a measure of change distribution. If a majority of files are changed in a version, it is likely that change propagated across functionalities. The assumption underlying this metric is that functions are grouped together in files based on the functionality they provide.
- *A-Tests-CCHF: average percentage of tests executing changed functions.* A-Tests-CCHF measures the average fraction of tests covering the changed functions. Once again, only changed functions that are covered by the test suite were considered. A high value for this metric is likely to reflect changes lying

on common test execution paths. In other words, it implies that functionality that is quite popular with the test suite has been changed. A low value for this metric means that the change is in rarely tested functionality.

- *P-CH-Index: probability of execution of changed functions.* P-CH-Index measures change “popularity”; its value is computed by summing the execution probabilities of changed functions. The execution probability of each function is computed by counting the execution frequency of each function when the test suite is executed. The value of this variable increases as more functions with high execution likelihood are changed.

### 3.2.2 Dependent variables

Three dependent variables were measured:

*Savings in number of test cases executed.* Regression test selection achieves savings by reducing the number of test cases that need to be executed on  $P'$ , thereby reducing the effort required to retest  $P'$ . To measure these savings, differences in the numbers of tests selected were measured.

*Savings due to increases in the rate of fault detection.* The test case prioritization techniques considered have a goal of increasing a test suite’s rate of fault detection. To measure rate of fault detection, [36] introduced a metric, *APFD*, which 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:

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

Examples and empirical results illustrating the use of this metric are provided in [8].

*Costs due to loss in fault detection effectiveness.* One potential cost of using regression test selection and test suite reduction techniques involves the costs of missing faults that would have been exposed by excluded test cases. When  $P'$  contains multiple faults, however, it is not sufficient to measure which test cases cause  $P'$  to fail; it is also important to determine which test cases could contribute to revealing which faults. To allow this determination, each fault  $f$  in  $P'$  was activated individually, executing each test case  $t$  on  $P'$ , and determined whether  $t$  detects  $f$ . It was assumed that detection of  $f$  when present singly implies detection of  $f$  when present in combination with other faults. Although this assumption may lead to overestimating fault detection in cases where multiple faults present together mask each other’s effects, a recent study on ten versions of two large subjects indicates that masking only occurs in an average of 1.6% of the test cases, providing a closer estimate than other methods used previously [21]. Further discussion of the tradeoffs involved in this approach can be found in [31].

### 3.3 Threats to Validity

Most previous studies on regression testing have been formal experiments, focusing on controlling factors that could affect results. Although that approach was appropriate when studying smaller programs, it does not scale easily to larger systems where sufficient control cannot be exercised over the factors that might affect the dependent variables. It is important, however, to consider larger programs, both to reduce threats to external validity and increase the extent to which results generalize. This work addresses threats to external validity by targeting software systems drawn from the field. The systems used are similar to a large class of practical systems. For example, the Linux RedHat 7.1 distribution includes source code for 394 applications; the average size of these applications is 22,104 executable lines of code, and 53% have sizes between 5 and 65 KLOC, which is the size range of the applications considered for this study. Different types of applications were also included: a Unix shell, a pattern searcher, a lexical analyzer generator and a compression utility. Still, identifying and preparing a valid sample of programs for this type of study is not an easy task and the results are affected by this sample.

A drawback of using case studies as an empirical approach is that, in general, they face stronger threats to internal validity than formal experiments. The lack of control on specific aspects of such studies generally increases the chance that unknown factors could affect results. In this study no control is imposed on the evolution of the programs. Each version contained certain changes that occurred during the program's maintenance cycle. If just one version of one program had been observed, then the resulting conclusions could have been biased due to the particular types of changes made in that version. To control for this threat, multiple versions of multiple programs were used.

Another internal validity threat involves the accuracy of the tools and processes used. Some of these processes involved programmers (e.g., fault seeding) and some of the tools were specifically developed for this study, all of which could have added variability to results. Several procedures were used to lessen these sources of variation. The fault seeding process was performed following a specification so that each programmer operated in a similar way, and it was performed in two locations using different groups of programmers. Also, new tools were validated by testing them on small sample programs and test suites, refining them as larger subjects were targeted, and cross-validating them between labs.

Another threat to internal validity was the availability of just one test suite for each program. Having just one test suite makes it difficult to determine the impact of test suite constitution on the results. On the other hand, the cost of preparing a test suite for any of these subjects is large (an average of 400 hours apiece for the smaller programs), which limits the number of test suites that can be prepared. To reduce this threat, several test suite attributes were measured during the analysis to determine if the results could be affected by it.

Next, consider threats to construct validity. The dependent measures that were considered in this study are not the only possible measures of the costs and benefits of regression testing methodologies. These measures ignore the human costs that can be involved in executing and managing test suites. These measures do not consider debugging costs such as the difficulty of fault localization, which could favor small granularity test suites [19]. These measures also ignore the analysis time required to select or prioritize test cases.

Previous work [34, 36] has shown, however, that for the techniques considered, analysis time is much smaller than test execution time, or analysis can be accomplished automatically in off-hours prior to the critical regression testing period. Last, the metrics selected to quantify the different dimensions of change might not be orthogonal since we have more metrics than dimensions. The existence of overlap among these metrics might indicate that further studies are necessary to refine the characterization of changes during software evolution.

Finally, consider threats to conclusion validity. The number of programs and versions considered might not have been large enough to detect other patterns (or to invalidate some of the patterns found). The use of additional versions would have increased confidence in the findings. However, the average cost of preparing each version exceeded 80 hours, limiting the possibilities for making additional observations.

### 3.4 Results and Observations

In this section, the results and observations for each of the units of analysis in the study are presented first individually, and then (Section 3.4.5) across those units. Later, Section 4 explores the practical implications of these observations, and relates them to the questions addressed.

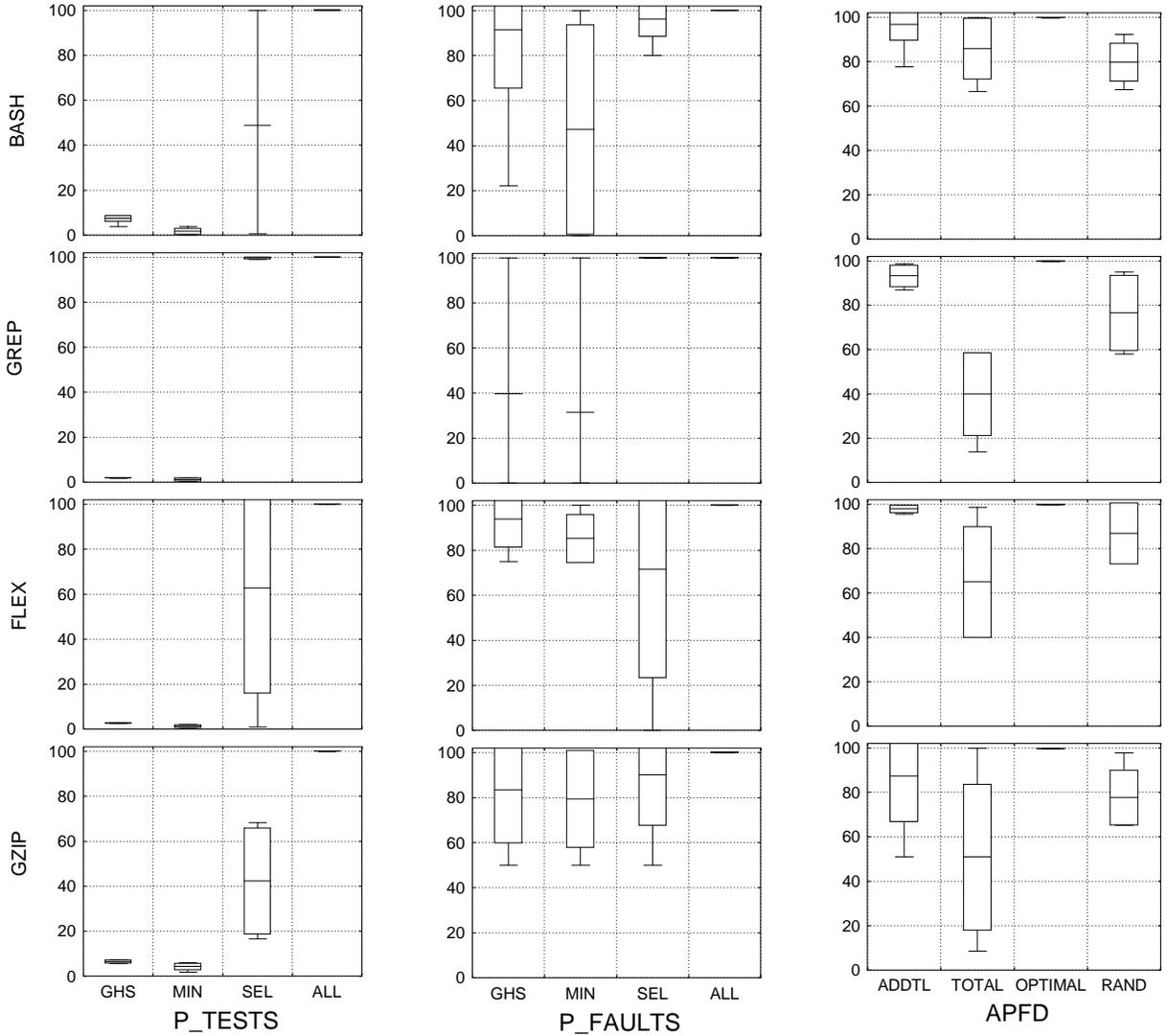
Each of the analyses that follow references the data shown in Table 4 and Figure 1. Table 4 presents the six independent change metrics collected, with their means and standard deviations, for each of the objects of study. Figure 1 displays the selection and prioritization results for each of the four observed programs. The results are presented through a series of twelve box plot diagrams (one per program and dependent variable). The diagrams show the distribution of the three dependent variables: percentage of test cases executed (P\_TESTS), percentage of faults detected (P\_FAULTS), and rate of fault detection (APFD), plotted separately for each regression test selection or prioritization technique. The mean is used as the measure of central tendency, the standard deviation is used to represent the variation that encloses each box, and the whiskers are used to represent the range of observed values.

#### 3.4.1 Bash

*Selection.* (Figure 1: row 1, columns 1 and 2). There is almost no variation in the percentage of test cases selected by coverage- and modification-focused minimization: each technique selected less than 10% of the test suite. However, coverage-focused minimization performed better than modification-focused minimization by detecting an average of 90% of the faults versus the 50% detected by the latter. Coverage-focused minimization detected all faults in eight out of the nine versions. This effectiveness could be attributed to the small number of lines changed (P-CH-L was less than 6% on all versions) across many functions and files (P-CH-F 13% and P-CH-Files 34%), which causes adequate coverage to be achieved quickly. Modification-focused minimization failed to detect faults on six of nine versions, and on three of those versions it missed all faults. All six versions on which faults were missed contained concentrated changes (four versions had less than 4% for P-CH-F) with a high percentage of test cases going through those changes (A-Tests-CCHF 67%). Since modification-focused minimization attempts to select a minimal set of test cases that exercises changed functions, it is likely to exclude some test cases that might expose the faults. This likelihood increases if the

	Bash		Grep		Flex		Gzip	
	mean	stdev	mean	stdev	mean	stdev	mean	stdev
P-CH-L	2.5	2.2	9.0	9.5	11.6	11.9	16.1	9.7
P-CH-F	12.9	10.6	45.4	33.0	26.2	22.5	33.9	14.0
A-CH-LOC-F	8.1	5.1	19.9	14.6	23.2	11.7	24.0	10.3
P-CH-Files	34.4	19.5	4.8	2.0	3.9	1.6	7.4	4.0
A-Tests-CCHF	69.3	8.6	39.7	3.8	32.6	3.1	6.2	1.1
P-CH-Index	13.8	14.4	49.3	32.6	28.1	22.6	25.2	8.6

Table 4: Basic statistics for change variables.



GHS: coverage-focused minimization, MIN: modification-focused minimization, SEL: modified non-core function selection, ALL: retest-all; ADDTL: additional coverage prioritization, TOTAL: total coverage prioritization, OPTIMAL: optimal prioritization, RAND: random ordering.

Figure 1: Distribution of dependent variables across versions, per program, per variable.

changes are executed by a large number of test cases.

Modified non-core function selection was the most consistent and the best performing test selection technique in terms of fault detection effectiveness. However, it did exhibit a large amount of variation in the number of test cases selected, ranging from less than 1% to 100% across versions. This variation seems to be tied directly to change distribution, as indicated by a 0.86 linear correlation (using Pearson's standard correlation) between the percentage of changed files (P-CH-Files) and the number of test cases selected.

*Prioritization.* (Figure 1: row 1, column 3). As expected, the optimal technique performed extremely well (APFD over 99% for all versions), with almost no variation in spite of the different size, distribution, and coverage of changes. The additional function coverage prioritization technique was second, achieving APFD above 96% on all but version 1. That particular version displayed a unique combination of change characteristics: a high percentage of changed functions (P-CH-F = 22%) covered by a high number of tests (A-Tests-CCHF = 74%). Since the additional function coverage prioritization technique assigns lower priority to test cases executing functions already covered, having a large number of tests traversing the changed (and possible faulty) functions was not beneficial. The total function coverage prioritization technique performance was consistently inferior to the additional function coverage technique, with APFD over 90% on only half of the versions. Furthermore, no obvious relationship could be observed between the change patterns and the performance of the total prioritization technique.

### 3.4.2 Grep

*Selection.* (Figure 1: row 2, columns 1 and 2). Coverage and modification-focused minimization selected less than 2% of the test cases on all versions. They also exhibited a large variation in percentage of faults detected, ranging from 0% to 100% (their boxes are not drawn because their standard deviation exceeded the margins of the graph). Both techniques performed best on version 4, where change was small (P-CH-L less than 3%), highly distributed across 90% of the functions with an average of three lines changed per function, and covered by the lowest percentage of tests in all versions. The other three versions contained relatively more concentrated changes and a larger number of changed lines of code per function (A-CH-LOC-F of 14, 25 and 37). Given the large distribution of change observed across all versions, modified non-core function selection always selected over 99% of the test cases which resulted in 100% fault detection, but with a cost similar to retest-all.

*Prioritization.* (Figure 1: row 2, column 3). Optimal prioritization performed as expected, that is, a high APFD and low variation. For this program, the additional function coverage technique achieved APFD over 86%. The total function coverage technique, on the other hand, presented varied results, for an APFD averaging only 40%. Change distribution clearly limited this technique's performance; for example, for version 4, 90% of the functions were changed, fewer than 3% of the lines of code were changed, and APFD was less than 14%.

### 3.4.3 Flex

*Selection.* (Figure 1: row 3, columns 1 and 2). Coverage-focused minimization was quite cost-effective, using less than 3% of the test suite while detecting all but one fault. The version with the missed fault had the highest P-CH-Index (0.6), which indicates that the changes were made to frequently executed functions. Since this technique eliminates test cases with redundant coverage, functions that are very “popular” might not receive all the testing they would under more conservative test selection techniques. Modification-focused minimization performed slightly better than coverage-focused minimization in terms of tests used (maximum of 2% in version 1), but with worse fault detection effectiveness, missing one fault in 75% of the versions. The versions in which faults were missed averaged 27 A-CH-LOC-F, while the version in which all faults were found had an A-CH-LOC-F of 9.0. Since modification-focused minimization selects one test case per changed function, having functions with multiple lines changed (and possible multiple locations where faults can reside) can jeopardize its effectiveness.

Surprisingly, modified non-core function selection displayed a great deal of variation in terms of both fault detection and test selection. However, on three of the versions, an average of 95% of the faults were detected. This variation was just due to the technique’s performance in version 4, where selection missed the only existing fault (resulting in 0% detected). That particular fault was located in one of the core functions that the selection technique ignored (a safe selection technique would have detected it).

*Prioritization.* (Figure 1: row 3, column 3). **Flex** displayed behavior similar to **grep**, with optimal and additional function coverage prioritization techniques always performing above 95%. The total function coverage technique again displayed a lot of variation. Version 4 of **flex** had an APFD of 98%, whereas the other 3 versions averaged an APFD of 53%. Interestingly, those three versions contained changes highly distributed across functions, and many lines of code changed within each changed function. For example, in version 2, 57% of the functions were changed and an average of 33 lines of code were changed per function, and here APFD was 42%. This confirms the previous observations that type of change can greatly limit the effectiveness of the total function coverage technique because prioritizing test cases based on the total coverage they provide might ignore the fact that changes (and their embedded faults) are in functions that are rarely covered.

### 3.4.4 Gzip

*Selection.* (Figure 1: row 4, columns 1 and 2). Coverage-focused minimization missed one fault in two versions, change-focused minimization missed one fault in three versions, and non-core modified function selection missed only one fault in one version. The number of faults detected for all techniques was lowest in version 3, which had the smallest amount of change (less than 4% of P-CH-L) and the most concentrated change (P-CH-F less than 14%) of all versions. Non-core modified function selection did select more test cases, averaging 42% across all versions. Again, non-core selection performed comparably to the other two techniques in terms of the percentage of faults detected for version 3, in which the number of functions

changed was the lowest.

*Prioritization.* (Figure 1: row 4, column 3). `Gzip` displayed the highest variation for the additional function coverage and total function coverage prioritization techniques among all subjects. Additional function coverage prioritization had APFD over 90% on four versions, and 51% on version 3. Version 3 contained the most concentrated changes (P-CH-F 13%) of all versions and the largest number of tests going through those changes (A-Tests-CCHF 8%), which plays against the “greedy” nature of the additional technique. The total function coverage prioritization technique achieved APFD under 60% for four versions. Three of those versions contained large and highly distributed changes, with an average of 30 lines of code changed per function, 41% of functions changed, and over 10% of the files affected by the changes.

### 3.4.5 Implications across case studies

The individual patterns described in the previous sections gain significance when observed repeatedly across multiple units of analysis. In this section, a more general data analysis is performed to offer a broader explanation about the sources of similarities and variation across programs, and the impact of change attributes on the cost-effectiveness of regression testing techniques.

*Implication 1: variation in change distribution and coverage affected the percentage of test cases selected under modified non-core function selection, but did not seem to affect test selection results for coverage- or modification-focused minimization.* Coverage- and modification-focused minimization provided substantial savings, selecting under 8% of the test cases on all versions of all programs. In contrast, for modified non-core function selection, the number of functions changed and the average number of test cases executing those functions greatly impacted the number of test cases selected. For `bash`, `flex`, and `gzip`, the correlation between P-CH-F and the number of test cases selected exceeded 0.75. For `grep`, which had the largest P-CH-F (45%), selection always exceeded 99% (this is compounded by the fact that `grep` had the largest P-CH-Index of all programs).

*Implication 2: small, highly distributed, and infrequently covered changes seem to benefit the fault detection effectiveness of coverage- and modification-focused minimization.* This explains the large variation in fault-detection effectiveness observed for these techniques on `grep`, for which one version had small but highly distributed changes and all faults were detected, while three versions had large numbers of lines of code changed per function, which clearly constrained these techniques’ performance. Also, modification-focused minimization performs consistently worse than coverage-focused minimization in terms of fault detection effectiveness when the changes were made to sections of code not frequently accessed by the test suite (as measured by P-CH-Index).

*Implication 3: the fault detection effectiveness of coverage- and modification-focused minimization is unpredictable in the presence of changes with high levels of coverage.* This explains the large variation observed for `bash`, where changed functions were executed by an average of 70% of the test cases. This trend is also evident in those versions of `grep` and `gzip` for which these techniques performed the worst. Also, fault detection effectiveness seems to decrease as the number of lines of code per function increases and the number of tests going through the changed functions increases.

*Implication 4: additional function coverage prioritization always performed well, but total function coverage prioritization was often unpredictable.* The additional function coverage technique, on average, always outperformed total function coverage and random ordering regardless of change characteristics, and also had low variation. For `flex` and `grep`, additional function coverage prioritization performed close to optimal prioritization on all versions. Total function coverage prioritization displayed large variation, with (for example) APFD ranging from nine to 100 on `gzip`, and performing worse than random prioritization on three of the four programs.

*Implication 5: highly distributed changes benefit additional function coverage prioritization, but can hurt total function coverage prioritization.* The difference in performance between the two techniques increases as changes are more distributed. `Grep`, `gzip`, `flex`, and `bash` have decreasing change distribution averages (see Table 4, second row), which matches the differences in performance between the two techniques. The two techniques also exhibit similar behavior if a large number of test cases execute the changed functions independently of change distribution. This is evident in all versions of `bash`, where almost 70% of the tests execute the changed functions.

## 4 Discussion and Conclusions

This article has presented the results of an embedded multiple case study of the impact of change patterns on the cost-effectiveness of several regression testing techniques. This is one of the first rigorous empirical studies of these issues, and it is also the first such study of these issues to use software drawn, with all of its releases, from the field, and from a pool of programs that represent an important class of software used in practice.

Overall, the results of this study confirm that change attributes play a significant role in the performance of regression testing techniques; this holds for all the heuristics that were investigated. More important from a practical perspective, however, are the ways in which engineers unaware of this role can make poor choices in designing regression test suites, building modifications into new releases of software, and selecting integration or regression testing strategies and techniques. Even more valuable are the ways in which knowledge of this role can enable better choices.

For example, change size does not seem to be the predominant factor in determining the cost-effectiveness of techniques. Instead, the distribution of changes across functions and files, and whether the test cases reached those changes, seem to be the main contributors to the variation observed across all programs. A simple lines-of-code mentality used to evaluate prospective modifications will not produce effective results.

Independent of change attributes, if practitioners are concerned primarily about test execution costs and are willing to sacrifice fault detection effectiveness, aggressively selective (non-conservative) test selection strategies, such as coverage-focused and modification-focused minimization, are sufficient in most scenarios. However, if practitioners are concerned about fault detection effectiveness while using these aggressive strategies, they should be advised to incorporate a minimal number of highly distributed changes into each tested build.

Integrating a large number of changes within a reduced number of functions, or even within a single fea-

ture, is likely to result in undetected faults under aggressively selective strategies. (In this context, “feature” means a distinguishable unit of functionality, such as may be captured by a use case, and is of interest due to the fact that features tend to be associated with clusters of similar paths through code.) It is worth also noting, however, that under these circumstances, coverage-focused minimization usually detects more faults than modification-focused minimization, because it includes some test cases that cover some unchanged functions that modification-focused minimization does not try to cover. These test cases may execute changed (and perhaps faulty) functions more times.

The practitioner should also know that if changes are incorporated into functions executed by multiple test cases, the fault detection effectiveness of coverage- and modification-focused minimization techniques is extremely difficult to predict, even when changes are highly concentrated. On the other hand, the same scenario of concentrated changes executed by a large percentage of test cases is ideal for conservative test selection techniques (like the modified non-core function technique), where it is likely to result in a small number of test cases selected and high fault detection effectiveness.

Another interesting observation is that, independent of test case execution patterns, if changes are concentrated, the size of the change will not affect conservative test selection techniques. Further, if the changes are distributed across several functions but still lie within a single feature, conservative test selection techniques continue to be very likely to produce savings in test execution, while still detecting the most faults. A quite different outcome becomes feasible, however, if a practitioner distributes multiple modifications across many functions and features. Under that scenario, conservative test selection is almost guaranteed to select all test cases, producing no savings. Under the same circumstances, employing the additional function coverage prioritization technique is highly recommended since it is almost guaranteed to perform close to optimal prioritization, discovering a large percentage of the faults early in the test cycle. On the other hand, total function coverage prioritization performs best if changes occur within the same feature, but even then its results are often discouraging.

Finally, moving onto more speculative implications, the observations drawn in this study might support the coupling of specific techniques with specific development approaches. The observation that conservative regression test selection techniques are most beneficial when changes are concentrated might provide an additional incentive for feature oriented programming [10], where development is particularly interested in mapping functional requirements to unique pieces of code. Along the same lines, the observations that small distributed changes are most amenable to the use of more aggressively selective techniques could justify its streamlined implementation in current open source development processes (e.g., the Mozilla project [25]), where tens of small fixes from different developers are integrated and tested nightly.

Speculation such as this, however, needs to be supported by additional empirical observation, and this is the focus of future work. This family of studies is being continued, with the aim of extending knowledge of which results generalize through case studies of additional subject systems, while extending precise knowledge of causal factors through controlled experiments.

## Acknowledgements

The authors thank Hyunsook Do, Amit Goel, Dalai Jin, Srikanth Karre, Khoa Le, and other members of the Galileo Research Group at Oregon State University and the Mapstext Group at University of Nebraska–Lincoln for their efforts in developing the infrastructure needed to support this study. The authors also thank Mary Jean Harrold and the Aristotle Research Group at Georgia Institute of Technology for providing access to the Aristotle tool suite. This work was supported in part by the NSF Information Technology Research program under Awards CCR-0080898 and CCR-0080900 to University of Nebraska, Lincoln and Oregon State University. The work was also supported in part by NSF Awards CCR-9703108 and CCR-9707792 to Oregon State University, respectively, and by an NSF-EPSCOR Grant Award to the University of Nebraska, Lincoln.

## References

- [1] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. In *Proceedings of the 3rd ACM Symposium on Software Testing, Analysis, and Verification*, pages 210–218, Dec. 1989.
- [2] T. Ball. On the limit of control flow analysis for regression test selection. In *Proceedings of the ACM International Symposium on Software Testing and Analysis*, pages 134–142, Mar. 1998.
- [3] J. Bible, G. Rothermel, and D. Rosenblum. Coarse- and fine-grained safe regression test selection. *ACM Transactions on Software Engineering and Methodologies*, 10(2):149–183, Apr. 2001.
- [4] Y. Chen, D. Rosenblum, and K. Vo. TestTube: A system for selective regression testing. In *Proceedings of the 16th International Conference on Software Engineering*, pages 211–220, May 1994.
- [5] 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, Apr. 2001.
- [6] 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, Aug. 2000.
- [7] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 329–338, May 2001.
- [8] 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, Feb. 2002.
- [9] S. Elbaum, J. Munson, and M. Harrison. CLIC: A tool for the measurement of software system dynamics. In *SETL Technical Report - TR-98-04.*, April 1998.

- [10] T. Elrad, R. Filman, and A. Bader. Article Series on Aspect Oriented Programming. *Communications of the ACM*, 44(10):29–97, 2001.
- [11] K. Fischer, F. Raji, and A. Chruscicki. A methodology for retesting modified software. In *Proceedings of the National Telecommunications Conference B-6-3*, pages 1–6, Nov. 1981.
- [12] T. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *Proceedings of the 20th International Conference on Software Engineering*, pages 188–197, Apr. 1998.
- [13] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):184–208, Apr. 2001.
- [14] R. Gupta, M. J. Harrold, and M. Soffa. An approach to regression testing using slicing. In *Proceedings of the Conference on Software Maintenance*, pages 299–308, Nov. 1992.
- [15] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.
- [16] M. J. Harrold, J. Jones, T. Li, S. Spoon, and A. Gujarathi. Regression test selection for Java software. In *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 312–326, Oct. 2001.
- [17] M. J. Harrold and G. Rothermel. Aristotle: A system for research on and development of program analysis based tools. Technical Report OSU-CISRC- 3/97-TR17, Ohio State University, Mar 1997.
- [18] J. Hartmann and D. Robson. Techniques for selective revalidation. *IEEE Software*, 16(1):31–38, Jan. 1990.
- [19] R. Hildebrandt and A. Zeller. Minimizing failure-inducing input. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 135–145, Aug. 2000.
- [20] 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.
- [21] J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test application frequency. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 126–135, June 2000.
- [22] H. Leung and L. White. Insights into regression testing. In *Proceedings of the Conference on Software Maintenance*, pages 60–69, Oct. 1989.
- [23] H. Leung and L. White. A study of integration testing and software regression at the integration level. In *Proceedings of the Conference on Software Maintenance*, pages 290–300, Nov. 1990.

- [24] H. Leung and L. White. A cost model to compare regression test strategies. In *Proceedings of the Conference on Software Maintenance*, pages 201–208, Oct. 1991.
- [25] Mozilla. Reference: Mozilla tinderbox framework. <http://tinderbox.mozilla.org>, 2002.
- [26] A. P. Nikora and J. C. Munson. Software evolution and the fault process. In *Twenty Third Annual Software Engineering Workshop, NASA/Goddard Space Flight Center*, page <http://sel.gsfc.nasa.gov/website/sew/1998/program.htm>.
- [27] 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 1998.
- [28] T. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.
- [29] C. Ramey and B. Fox. *Bash Reference Manual*. O’Reilly & Associates, Inc., Sebastopol, CA, 2.2 edition, 1998.
- [30] D. Rosenblum and E. J. Weyuker. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Transactions on Software Engineering*, 23(3):146–156, Mar. 1997.
- [31] 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 24th International Conference on Software Engineering*, pages 230–240, May 2002.
- [32] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, Aug. 1996.
- [33] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering Methodology*, 6(2):173–210, Apr. 1997.
- [34] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24(6):401–419, June 1998.
- [35] G. Rothermel, M. J. Harrold, and J. Dedhia. Regression test selection for C++ programs. *Journal of Software Testing, Verification, and Reliability*, 10(2):77–109, June 2000.
- [36] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Test case prioritization. *IEEE Transactions on Software Engineering*, 27(10):929–948, Oct. 2001.
- [37] W. Wong, J. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of the Eighth International Symposium Software Reliability Engineering*, pages 230–238, Nov. 1997.
- [38] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set size and block coverage on the fault detection effectiveness. In *Proceedings of the Fifth International Symposium on Software Reliability Engineering*, pages 230–238, Nov. 1994.

- [39] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *Proceedings of the 17th International Conference on Software Engineering*, pages 41–50, Apr. 1995.
- [40] R. K. Yin. *Case Study Research : Design and Methods (Applied Social Research Methods, Vol. 5)*. Sage Publications, London, UK, 1994.