An Empirical Study of Regression Test Selection Techniques

Todd L. Graves*
Mary Jean Harrold†
Jung-Min Kim‡
Adam Porter§
Gregg Rothermel¶

ABSTRACT

Regression testing is an expensive maintenance process directed at validating modified software. Regression test selection techniques attempt to reduce the cost of regression testing by selecting tests from a program's existing test suite. Many regression test selection techniques have been proposed. Although there have been some analytical and empirical evaluations of individual techniques, to our knowledge only one comparative study, focusing on one aspect of two of these techniques, has been performed. We conducted an experiment to examine the relative costs and benefits of several regression test selection techniques. The experiment examined five techniques for reusing tests, focusing on their relative abilities to reduce regression testing effort and uncover faults in modified programs. Our results highlight several differences between the techniques, and expose essential tradeoffs that should be considered when choosing a technique for practical application.

KEYWORDS

regression testing, selective retest, empirical study

1 INTRODUCTION

As developers maintain a software system, they periodically *regression test* it, hoping to find errors caused by their changes. To do this, developers often create an initial test suite, and then reuse it for regression testing.

*National Institute of Statistical Sciences, and Software Production Research Department, Bell Laboratories, 1000 E. Warrenville Rd., Naperville, IL 60566, graves@bell-labs.com.

†Department of Computer and Information Science, Ohio State University, harrold@cis.ohio-state.edu.

‡Department of Computer Science, University of Maryland at College Park, jmkim@cs.umd.edu.

§Department of Computer Science, University of Maryland at College Park, aporter@cs.umd.edu.

¶Department of Computer Science, Oregon State University, grother@cs.orst.edu.

The simplest regression testing strategy, *retest all*, reruns every test in the initial test suite. This approach, however, can be prohibitively expensive – rerunning all tests in the test suite may require an unacceptable amount of time. An alternative approach, *regression test selection*, reruns only a subset of the initial test suite. Of course, this approach is imperfect as well – test selection techniques can have substantial costs, and can discard tests that could reveal faults, possibly reducing fault detection effectiveness.

This tradeoff between the time required to select and run tests and the fault detection ability of the tests that are run is central to regression test selection. Because there are many ways in which to approach this tradeoff, a number of test selection techniques have been proposed (e.g., [1, 4, 8, 9, 12, 15, 21]). Although there have been some analytical and empirical evaluations of individual techniques [4, 18, 20, 21], to our knowledge only one comparative study, focusing on one aspect of two of these techniques, has been performed [16].

We hypothesize that different regression test selection techniques create different tradeoffs between the costs of selecting and executing tests, and the need to achieve sufficient fault detection ability. Because there have been few controlled experiments to quantify these tradeoffs, we conducted such a study. Our results indicate that the choice of regression test selection algorithm significantly affects the cost-effectiveness of regression testing. Below we review the relevant literature, describe the test selection methods we examined, and present our experimental design, analysis, and conclusions.

2 REGRESSION TESTING SUMMARY AND LITERATURE REVIEW

2.1 Regression Testing

Let P be a procedure or program, let P' be a modified version of P, and let T be a *test suite* for P. A typical regression test proceeds as follows:

1. Select $T' \subseteq T$, a set of tests to execute on P'.
2. Test P' with T', establishing P''s correctness with respect to T'.

3. If necessary, create T'', a set of new functional or structural tests for P'.
4. Test P' with T'', establishing P''s correctness with respect to T''.
5. Create T''', a new test suite and test history for P', from T, T', and T''.

Although each of these steps involves important problems, in this article we restrict our attention to step 1, which involves the *regression test selection problem.*

2.2 Regression Test Selection Techniques

A variety of regression test selection techniques have been described in the research literature. Rothermel and Harrold [20] describe several families of techniques; we consider three such families, along with two additional approaches often used in practice. We here describe these families and approaches, and give a representative example of each – we utilize these representative examples in our experimentation.

2.2.1 Minimization Techniques. These techniques (e.g., [5, 9]) attempt to select minimal sets of tests from T, that yield coverage of modified or affected portions of P. One such technique requires that every program statement added to or modified for P' be executed (if possible) by at least one test in T.

2.2.2 Safe Techniques. These techniques (e.g., [4, 11, 21]) select, under certain conditions, every test in T that can expose one or more faults in P'. One such technique selects every test in T that, when executed on P, exercised at least one statement that has been deleted from P, or at least one statement that is new in or modified for P'.

2.2.3 Dataflow-Coverage-Based Techniques. These techniques (e.g., [8, 15, 22]) select tests that exercise data interactions that have been affected by modifications. One such technique selects every test in T that, when executed on P, exercised at least one definition-use pair that has been deleted from P', or at least one definition-use pair that has been modified for P'.

2.2.4 Ad Hoc / Random Techniques. When time constraints prohibit the use of a retest-all approach, but no test selection tool is available, developers often select tests based on "hunches", or loose associations of tests with functionality. One simple technique randomly selects a predetermined number of tests from T.

2.2.5 Retest-All Technique. This techniques reuses all existing tests. To test P', the technique "selects" all tests in T.

2.3 Previous Empirical Work

Unless test selection, program execution with the selected tests, and validation of the results take less time than rerunning all tests, test selection will be impractical. Therefore, cost-effectiveness is one of the first questions researchers in this area have studied.

Rosenblum and Weyuker [17, 18] and Rothermel and Harrold [21] conducted empirical studies to determine whether certain safe regression testing techniques are cost-effective relative to retest all.

Rosenblum and Weyuker applied their regression test selection algorithm, implemented in a tool called `TestTube`, to 31 versions of the KornShell and its associated test suites. For 80% of the versions, their algorithm required 100% of the tests. The authors note, however, that the test suite for KornShell contained a relatively small number (16) of test cases, many of which caused all components of the system to be exercised.

In contrast, Rothermel and Harrold applied their regression test selection algorithm, implemented in a tool called `DejaVu`, to a variety of 100-500 line programs, for which savings averaged 45%, and to a larger (50,000 line) software system, for which savings averaged 95%.

Thus, although our understanding of the issue is incomplete, there is some evidence to suggest that test selection can provide savings. Thus, further empirical investigation of test selection is warranted.

To our knowledge, the only existing comparative study of regression test selection techniques [16], by Rosenblum and Rothermel, compared the test selection results of `TestTube` and `DejaVu`. The study showed that `TestTube` was frequently competitive with `DejaVu` in terms of its ability to reduce the number of test cases selected, but that `DejaVu` sometimes substantially outperformed `TestTube`. The study did not consider relative fault-detection abilities, or compare techniques other than safe techniques.

2.4 Open Questions

None of the studies just described examined non-safe techniques, and none compared more than two techniques. Because non-safe techniques can discard *fault-revealing* tests (test that reveal faults in the modified program), the tradeoffs between test selection and fault detection should be explored, and techniques compared.

Several questions arise when we compare safe and non-safe techniques:

- How do techniques differ in terms of their ability to reduce regression testing costs?
- How do techniques differ in terms of their ability to detect faults?
- What tradeoffs exist between test suite size reduction and fault detection ability?
- When is one technique more cost-effective than another?

- How do factors such as program design, location and type of modifications, and test suite design affect the efficiency and effectiveness of test selection techniques?

3 THE EXPERIMENT

3.1 Hypotheses

H1: Safe techniques are more effective than minimization techniques, but are much more expensive.

H2: Dataflow-coverage-based techniques are nearly as effective as safe techniques, but can be more expensive.

H3: Dataflow-coverage-based techniques are more effective than minimization techniques, but are much more expensive.

H4: Non-random techniques are more effective than random techniques, but are much more expensive.

H5: The composition of the original test suite greatly affects the costs and benefits of test selection techniques.

3.2 Operational Model

To test our hypotheses we needed to measure the costs and benefits of each test selection algorithm. To do this we constructed two models: one for calculating the cost of using a test selection technique, and another for calculating the fault detection effectiveness of the resulting test suite. We here restrict our attention to the costs and benefits defined by these models, but there are many other costs and benefits these models do not capture. Some possible additions to the models are discussed in Section 5.

3.2.1 Modeling Cost-Effectiveness. Leung and White [13] present a cost model for selective retest techniques. Their model considers both test selection and identification of inadequately tested components; we adapt it to consider just the cost-effectiveness of a regression test selection technique relative to that of the retest-all approach.

In our model, the cost of regression test selection is $A + E(T')$, where A is the cost of the analysis required to select tests and $E(T')$ is the cost of executing and validating the selected tests. The cost of the retest-all technique is $E(T)$, where $E(T)$ is the cost of executing and validating all of the tests.

This model makes several simplifying assumptions. It assumes that the cost of executing tests is the same under test selection and retest all, and that all test cases have uniform costs [13]. It also assumes that all subcosts can be expressed in equivalent units, whereas, in practice, they are often a mixture of CPU time, human effort, and equipment costs [18].

Given this model, we needed to measure two things: the reduction in the cost of executing and validating tests, and the average analysis cost. Given our assumptions, however, the former can be measured in terms of test suite size reduction, as $(\frac{|T'|}{|T|})$. For several reasons we did not measure analysis costs directly. Most importantly, we did not possess implementations of all techniques, and instead were required to simulate techniques for which we had no implementations. Furthermore, because the experimental design required us to run over 185,000 test suites, we utilized several machines. We did not believe that the performance metrics taken from different machines were comparable. Instead, as we show in Section 4.4, we try to identify how large analysis costs can be before they outweigh reductions in test suite size.

3.2.2 Modeling Fault-Detection Effectiveness. Test selection techniques attempt to lower costs by running a subset of an existing test suite, but this approach may allow some fault-revealing tests to be discarded. Because an important benefit of testing is that it detects defects, it is important to understand whether, and to what extent, test selection reduces fault detection. We considered two methods for calculating reductions in fault detection effectiveness.

On a per-test basis: One way to measure a reduction in the fault-detection effectiveness of a selective retest technique, given program P and faulty version P', is to identify those tests that are in T and reveal a fault in P', but that are not in T'. This quantity can then be normalized by the number of fault-revealing tests in T. One problem with this approach is that multiple tests may reveal a given fault. In this case some tests could be discarded without reducing effectiveness; however, this measure penalizes such a decision.

On a per-test-suite basis: Another approach is to classify the results of test selection into one of three outcomes: (1) no test in T is fault-revealing, and, thus, no test in T' is fault-revealing; (2) some test in both T and T' is fault-revealing; or (3) some test in T is fault-revealing, but no test in T' is fault-revealing. Outcome 1 denotes situations in which the test suite is inadequate. Outcome 2 indicates test selection that does not reduce fault detection, and Outcome 3 captures those times in which test selection compromises fault detection.

We selected the second method for use in our analysis. Under this approach, for each program, our measure of fault detection effectiveness is the percentage of cases in which T' contains no fault-revealing tests, (i.e., outcome 3 occurs.)

3.3 Experimental Instrumentation

3.3.1 Programs. For our study, we obtained six C programs, with a number of modified versions and test

Program	*Functions*	*Lines*	*Versions*	*Avg T Size*
replace	21	516	32	398
printtokens2	19	483	10	389
schedule2	16	297	10	234
schedule	18	299	9	225
totinfo	7	346	23	199
tcas	9	138	41	83

Table 1: Experimental Subjects.

suites for those programs. The subjects were collected and constructed initially by Hutchins et al. [10] for use in experiments with dataflow- and controlflow-based test adequacy criteria; we modified some of the programs and versions slightly to enable their use with our tools. Table 1 describes the subjects. The programs range in size from 8 to 21 functions, and from 138 to 516 lines of code. We describe the other data in the table in the following paragraphs.

3.3.2 Tests, Test Pools, Versions, and Test Suites. Hutchins et al. constructed tests for these programs by a process described in [10]. For each base program, they created a *test pool* of black-box tests, using the *category partition method* and Siemens Test Specification Language tool [2, 14]. They then created additional white-box tests by hand and added them to the test pool, to ensure that each exercisable statement, edge, and definition-use pair in the base program or its control flow graph was exercised by at least 30 tests.

Hutchins et al. created *faulty* modified versions of base programs by manually modifying code in the base version; in most cases they modified a single line of code, and in a few cases they modified between 2 and 5 lines of code. Their goal was to introduce faults that were as "realistic" as possible, based on their experience with real programs. To obtain meaningful results, they retained only faults that were detectable by at least three and at most 350 tests in the associated test pool. For regression testing experiments, we consider the faulty modified versions of base programs to be ill-fated attempts to create modified versions of the base programs. We can then study the relative effectiveness of various test selection techniques at detecting the faults.

We used test pools to obtain two sets of test suites for each program: edge-coverage-based and non-coverage-based. To obtain edge-coverage-based test-suites, we used the test pools for the base programs, and test coverage information that we gathered for the tests, to generate 1000 edge-coverage-adequate test suites for each program. More precisely, to generate a test suite T for base program P from test pool T_p, we considered each edge in the control flow graph G for P. For each such edge E, we obtained a list of tests $T_p(E) \subseteq T_p$ that had exercised that edge. We then used the C pseudo-random-number generator "rand", seeded initially with the output of the C "time" system call, to obtain an integer which we treated as an index i into $T_p(E)$ (modulo $|T_p(E)|$). We added test i from $T_p(E)$ to T if it was not already present in T. Table 1 lists the average sizes of the test suites that we generated.

For each program, we also generated 1000 non-coverage-based test suites. To generate the kth non-coverage-based test suite T for base program P ($1 \leq k \leq 1000$), we determined n, the number of tests in the kth coverage-based test suite, and then chose tests randomly from the test pool for P and added them to T until T contained n tests. This process yielded non-coverage-based test suites of the same size as the coverage-based suites.

3.3.3 Test Selection Tools. To perform the experiments, we required implementations or simulations of several regression test selection tools. As a *minimization* technique, we created a simulator tool that selects a minimal set of tests T' such that T' is edge-coverage-adequate for edges in the control flow graphs for P or P' that have been modified. As a *safe* technique, we utilized an implementation of Rothermel and Harrold's regression test selection algorithm, implemented as a tool called `DejaVu`, and available as a component of the `Aristotle` program analysis system: the system and tool are described in [7, 19]. As a *dataflow-coverage-based technique*, we simulated a tool by manually inspecting program modifications, and generating a list of tuples that represent the definition-use pairs that are affected by the modifications. We then used a data-flow testing tool [6] to identify the tests in the test suite that satisfy the definition-use pairs. As a *random(n) technique* we created a tool that randomly selects n% of the tests from the suite. The *retest-all technique* required no implementation.

3.4 Experimental Design

3.4.1 Variables. The experiment manipulated three independent variables:

1. the subject program (6 programs, each with a variety of modified versions).
2. the test selection technique (safe, dataflow, minimization, random(25), random(50), random(75), retest all); and
3. test suite composition (edge-coverage-adequate or random).

On each run, with program P, version P', technique M, and test suite T, we measured:

1. the ratio of tests in the selected test suite T' to tests in the original test suite; and

2. whether one or more tests in T' reveals the fault in P'.

Each combination of program, version, and technique had 100 associated test suites. From these 100 data points we computed two dependent variables.

1. average reduction in test suite size, and
2. fault detection effectiveness (the percentage of test suites in which T' does not reveal a fault in P'.)

3.4.2 Design. This experiment uses a full-factorial design with 100 repeated measures. That is, for each subject program, we selected 100 edge-coverage-adequate and 100 random test suites from the test-suite universe. For each test suite, we then applied each test selection method and evaluated the fault detection effectiveness of the resulting test suites.

3.4.3 Threats to Internal Validity. Threats to internal validity are influences that can affect the dependent variables without the researcher's knowledge. Our greatest concern is instrumentation effects that can bias our results.

Instrumentation effects are caused by differences in the test process inputs: the code to be tested, the locality of the program change, or the composition of the test suite. In this study, we use two different criteria for composing test suites: one in which test suites are randomly selected from the test pool, and one in which the test suite must provide edge coverage. However, at this time we do not control for the structure of the subject programs, nor for the locality of program changes. To limit problems related to this, we run each test selection algorithm on each test suite and each subject program.

3.4.4 Threats to External Validity. Threats to external validity are conditions that limit our ability to generalize the results of our experiment to industrial practice. We considered two sources of such threats: (1) artifact representativeness, and (2) process representativeness.

Artifact representativeness is a threat when the subject programs are not representative of programs found in industrial practice. There are several such threats in this experiment. First, the subject programs are of small size. As discussed earlier, there is some evidence to suggest that larger programs allow greater test set reduction, although at higher cost than small programs. Thus, larger program may be subject to different cost-benefit tradeoffs. Also, there is exactly one seeded error in every subject program. Industrial programs have much more complex error patterns.

Threats regarding process representativeness arise when the testing process we use is not representative of the industrial one. This may also endanger our results because our test suites may be more or less comprehensive than those created in practice. Also, our experiment mimics a corrective maintenance process, but there are many other times in which regression testing might be used.

3.4.5 Threats to Construct Validity. Threats to construct validity arise when measurement instruments do not adequately capture the concepts they are supposed to measure. For example, in this experiment our measures of cost and effectiveness are very coarse. For example, they treat all faults as equally severe.

3.4.6 Analysis Strategy. Our analysis strategy has three steps. First we summarize the data. Then we compare the ability of the test selection methods to reduce test suite size, and we compare the fault-detection effectiveness of the resulting test suites. Here, we establish that, in general, larger reductions in test suite size lead to greater reductions in fault detection effectiveness. Finally, we make several comparisons between nonrandom (i.e., minimization, safe, and dataflow) and random methods. For example, in one analysis we explore how large analysis costs can become before the non-random methods become less cost-effective than random ones.

4 DATA AND ANALYSIS

Two sets of data are important for this study: the test selection and the fault-detection summaries. This information is captured for every test suite, every subject program, and every test selection method. The test selection summary gives the size (in number of tests) of T and T'. From this information we calculate the percentage reduction in test suite size. The fault-detection summary shows whether T and T' contain any fault-revealing tests. From this information we determine whether the test selection technique compromised fault-detection effectiveness.[1]

In this article, we frequently use box plots (e.g., Figure 1) to represent data distributions. In these plots, a box represents each distribution. The box's height spans the central 50% of the data and its upper and lower ends mark the upper and lower quartiles. The bold dot within the box denotes the median. The dashed vertical lines attached to the box indicate the tails of the distribution; they extend to the standard range of the data (1.5 times the inter-quartile range). All other detached points are "outliers". We also use arrays of boxplots (a type of Trellis display [3]) to show data distributions that are conditioned on one or more other variables (e.g., Figure 2.)

By conditioned, we mean that data are partitioned into subsets, such that the data in each subset have the same

[1]Readers who wish to examine the data should contact Adam Porter.

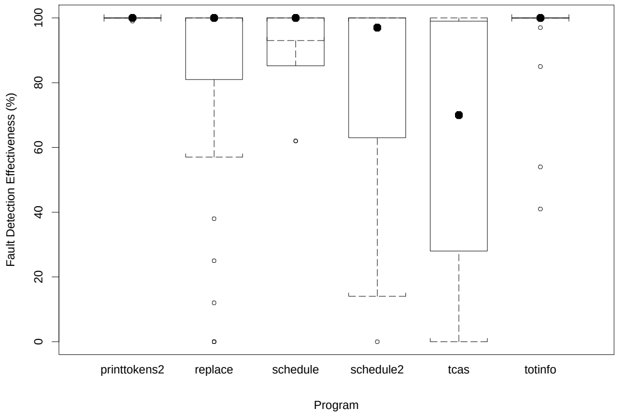

Figure 1: Fault detection effectiveness by program.

value for the conditioning variable. For example Figure 3 depicts the fault detection effectiveness for test suites created by different methods, conditioned on the program on which the test suite was run. That means that the data is partitioned into six subsets; one for each program. And then we draw one boxplot for each subset.

4.1 Test Suite Characteristics

Table 1 provides the average test suite sizes for the six programs. We show the data from the coverage-based suites only, because by construction, the random suites have the same sizes. The average test suite size ranges from about 100 tests for *tcas* to about 400 for *replace*.

Figure 1 depicts the distribution of fault detection effectiveness for all test suites over the six programs. We see that effectiveness differs substantially across different programs. For example, the median fault detection effectiveness for *tcas* is only about 70%. We can also see that some combinations of test suites and program versions have an effectiveness at or near 0 (e.g., both *replace* and *schedule2* exhibit outliers at 0%).

4.2 Test Suite Size Reduction

Figure 2 depicts the ability of each method to reduce test suite size, conditioned on program. For these programs, we see that the random methods extract a constant percentage of the tests (by construction) and that minimization (by nature of the modifications made to the subjects) almost always (91%) selects only 1 test. Interestingly, the safe and the dataflow methods have nearly identical performance (median reduced suite size is 74% for coverage suites and 58% for random).

4.3 Fault Detection

Figure 3 depicts the fault-detection effectiveness of test suites selected with each method, conditioned on program. Overall, we found that minimization had the lowest fault-detection effectiveness. The effectiveness of the random methods increased with test suite size, but that rate of increase diminished as size increased. Again the safe and dataflow methods exhibited similar median performances, but the dataflow distribution exhibited greater variance. This occurs because in some cases dataflow allows faults to escape, while the safe method does not.

4.4 Cost-Benefit Tradeoffs

We find a clear tradeoff between test suite size and detection rate. As the size of the selected test suite decreases so does fault-detection effectiveness. This relationship is depicted in Figure 4.

If we do not consider the analysis costs of non-random methods, then the decision to use a particular selection method will depend on the penalty for missing faults in relation to the cost of running more tests. This will obviously depend on many context-specific factors.

In this section we explore the effect of analysis costs for non-random methods on the relationships in Figure 4. To do this we examine how each non-random method compares to random methods and to each other. We assume that the analysis costs for non-random methods can be stated in terms of the cost to run a single test (analysis costs for random methods are 0), and then we characterize how many tests can be run (i.e., how long analysis can take) before the non-random method becomes less cost-effective than random ones.

We begin with minimization, the rule with the smallest test suites and lowest fault-detection effectiveness. We will compare its detection rate to that of a randomized rule calibrated to have the same total computational cost. Our goal is to find an upper bound, k, on the

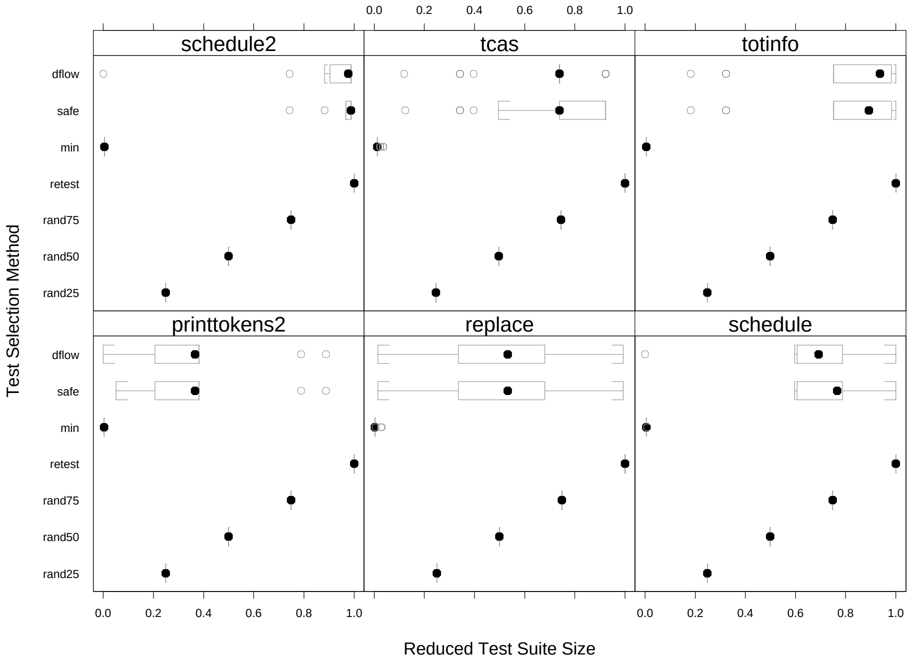

Figure 2: Test suite reduction by method, conditioned on program.

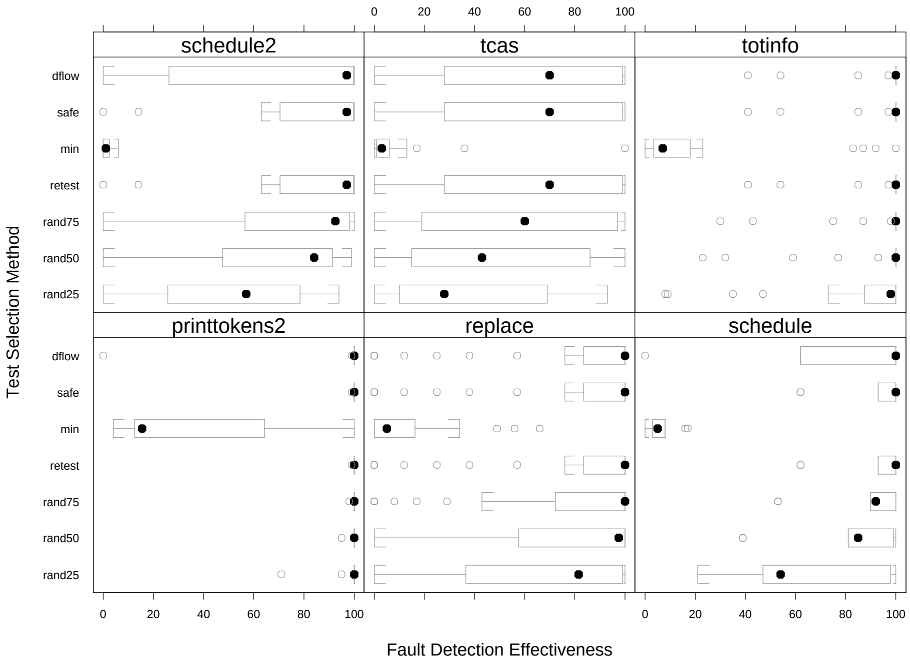

Figure 3: Fault-detection effectiveness by selection method, conditioned on program.

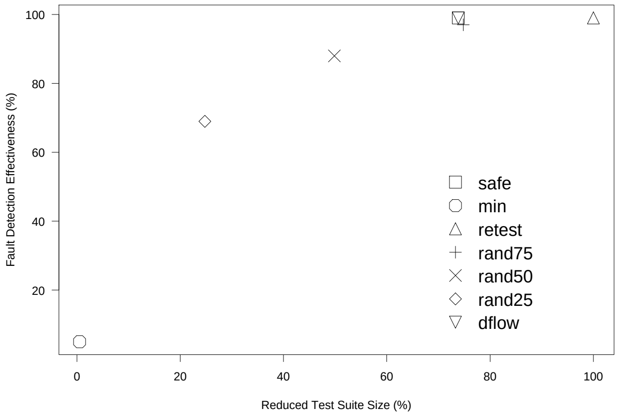

Figure 4: Fault-detection effectiveness and test suite size, irrespective of analysis costs.

analysis cost of minimization. That is, if the analysis costs are greater than the cost of running k tests, then there exists a random method that is less expensive and has the same fault-detection effectiveness.

We then perform similar analyses comparing the safe and the dataflow method to other randomized rules, retest all, and to each other. After each comparison we discuss our interpretations and their limitations.

4.4.1 Minimization vs. Randomization. The test suites selected by minimization were both the smallest (median of 1 test) and the least effective ($< 0.05\%$). (Of course, if the programs contained multiple changes, larger minimization suites would be selected.) Other methods typically selected test suites with on the order of 100 tests, and were much more effective. Because minimization does choose very few tests, there may be situations in which its use is appropriate, namely when tests are very expensive to run and missed faults are not considered excessively costly. Therefore, further study is warranted. In particular, we are also interested in knowing how much analysis cost minimization can incur before a random method would be preferable.

In this analysis we assume that a method's analysis time is equivalent to the cost of running k tests. We then determine a critical value of k for which there is a random reduction rule whose performance is as good or better than minimization's. If analysis costs exceed this critical value, then a random reduction rule may be more cost-effective.

Ideally we would like to compare minimization to a rule that chooses $100p\%$ tests at random, where this is equal to the average size of minimization test suites. In our experiment we only constructed random test suites with $p \in \{0.25, 0.5, 0.75, 1\}$ (and, in effect, $p = 0$). So we simulate the long-run behavior of an arbitrarily-sized random method by randomizing over values of p for which we have test suites. For instance, if we want to simulate random(5), we use random(25) with probability 0.2, and do no regression testing at all (random(0)) with probability 0.8 (our experiments suggest that this approach underestimates the effectiveness of the true random method, and, thus, overestimates the value of k we are looking for).[2]

For a fixed trial value of k, and program version, we computed the average test suite size using minimization (call this x). We then used either random(25) or random(0), with the distribution chosen to ensure that the average size of these test suites was $x + k$. We then compared the detection effectiveness of the two methods. We continued to adjust k until the detection effectiveness was equal.

We found that for $k = 2.5$, the randomized rule had higher detection rates in 62 program-versions, minimization had higher detection rates in 58 program-versions, and there were 5 program-versions where neither scheme ever found the fault. (These results are calculated with respect to the coverage-based test suites. For random test suites, k was 2, so that random test suites are perhaps slightly more unfavorable to minimization.)

4.4.2 Safe vs. Dataflow. In our experiment, the safe and dataflow methods behaved very similarly. Their fault-detection performances were nearly identical, and for many versions they chose the same average numbers of tests per test suite. In all but three versions, both methods selected test suites equivalent in their ability

[2]Note that our simulation is not a practical selection rule because it assumes that we know a priori how many tests will be selected. Nevertheless, it does provide a measure of the usefulness of test selection algorithms.

to reveal fault-revealing tests. Consequently, the results for the safe and dataflow techniques agree so closely that the comparisons of safe with randomized methods and "retest all" in the following sections are nearly identical to the results for dataflow. However, dataflow coverage techniques may require more analysis time than safe techniques.

4.4.3 Safe vs. Randomization. The analysis here is similar to the previous analyses, except that the safe method always found the fault if a fault-revealing test existed. Therefore no random method has the same detection effectiveness as the safe method. Instead, we look for random methods that found a fixed percentage ($100(1-p)\%$) of the faults. Then, we again determine a value of k, such that there is a randomized method with the same total cost as the safe method and $100(1-p)\%$ the detection effectiveness.

We found that there exists a randomized rule with the same average test suite size (i.e., k=0) as the safe method that finds faults 93% ($p = 0.07$) as often in half the program-versions as the safe method does. When $k = 0.5$ there is a randomization rule as costly as the safe method that detects faults 95% as often in half the program-versions. When $k = 6$, the randomization rule detects faults at least 98% as often as the safe method in half the program-versions.

4.4.4 Safe vs. Retest all. The safe method always found all faults that could be found given the test suites used. Therefore, a safe method is preferable to running all tests in the test suite if and only if analysis costs are less than the costs of running the unselected tests. Figure 2 contains data showing the sizes of test suites selected by the safe method. It demonstrates that test suite reduction depends dramatically on the program: selected test suites for schedule2 were typically 99% as large as the original suites, while those for printtokens2 are about 37% as large.

5 SUMMARY AND CONCLUSIONS

In this article we present initial results of an empirical study of selective regression techniques. This study examined some of the costs and benefits of several test selection methods. Our results, although preliminary, highlight several differences among the techniques, expose essential tradeoffs, and provide an infrastructure for further research by ourselves and others.

As we discussed earlier, this experiment, like any other, has several limits to its validity. Keeping this in mind, we drew several observations from this work.

- Minimization produced the smallest and the least effective test suites. Although fault detection is obviously important, there are cases where testing is very expensive. In these cases minimization may be cost-effective. Nevertheless, for the programs and test suites we studied, random selection of just slightly larger suites (only 2.5 more tests) yielded equally fault detection results equivalent to those of minimization (on average) with none of the analysis costs. One limitation here is that "on the average" applies to long-run behavior. Half of the time the random method was as effective as minimization, half of the time it was not. If greater confidence is required, then the random methods will need to select more than 2.5 additional tests. Another limitation is that, in practice, one cannot know how many tests minimization (or any other test selection algorithm) would pick without actually running it.

- The safe and dataflow methods had nearly equivalent average behavior in terms of cost-effectiveness, typically detecting the same faults, and selecting the same size test suites. However, because dataflow-coverage-based techniques require at least as much analysis as the two most efficient safe techniques [4, 21], we saw little reason to recommend dataflow if test selection alone is the goal. However, dataflow techniques can be useful in other parts of regression testing, such as in identification of portions of T' that are not adequately tested by T. In other words, our model does not capture all possible costs or benefits of selective regression testing, and thus, may be too coarse for some situations.

- The safe method found all faults for which we had fault-revealing tests while selecting 74% of the tests on the average. However, we saw that for several programs it could not reduce the test suites at all. Also, we found that, on the average, only slightly larger random test suites could be nearly as effective. Again, we have to remember that we are making a probabilistic assessment. This raises an important measurement question. That is, when should we analyze methods like these on a case by case basis, and when is an amortized analysis more appropriate.

- We found that our results were sensitive not only to the selection methods we used, but also to the programs, the characteristics of the changes, and the composition of the test suites. We believe that it important to understand more precisely how these factors affect our methods. Without this information, we may mistake the effect of a non-representative workload for differences in methods.

We are continuing this family of experiments. In the future, we plan to (1) improve our cost models to include factors such as testing overhead and to better handle analysis costs, (2) extend our analysis to multiple types of faults, (3) develop time-series-based models, capturing notions of amortized analysis and non-constant fault densities, and (4) rerun these experiment using larger programs with more complex fault distributions.

6 ACKNOWLEDGMENTS

This work was supported in part by grants from Microsoft, Inc. to Ohio State University and Oregon State University, by National Science Foundation National Young Investigator Award CCR-9696157 to Ohio State University, by National Science Foundation Faculty Early Career Development Award CCR-9501354 to University of Maryland, by National Science Foundation Faculty Early Career Development Award CCR-9703108 to Oregon State University, by National Science Foundation Award CCR-9707792 to Ohio State University, University of Maryland, and Oregon State University, by National Science Foundation Grant SBR-9529926 to the National Institute of Statistical Sciences, and by an Ohio State University Research Foundation Seed Grant. Rui Wu collected some of the data for the data-flow testing experiments. Siemens Laboratories supplied the subject programs.

REFERENCES

[1] H. Agrawal, J. Horgan, E. Krauser, and S. London. Incremental regression testing. In *Proc. of the Conf. on Softw. Maint.*, pages 348–357, Sept. 1993.

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

[3] J. M. Chambers, W. S. Cleveland, B. Kleiner, and P. A. Tukey. *Graphical Methods for Data Analysis.* Wadsworth Int'l. Group, Belmont, CA, 1983.

[4] Y. Chen, D. Rosenblum, and K. Vo. TestTube: A system for selective regression testing. In *Proc. of the 16th Int'l. Conf. on Softw. Eng.*, pages 211–222, May 1994.

[5] K. Fischer, F. Raji, and A. Chruscicki. A methodology for retesting modified software. In *Proc. of the Nat'l. Tele. Conf. B-6-3*, pages 1–6, Nov. 1981.

[6] M. Harrold, J. A. Jones, and J. Lloyd. Design and implementation of an interprocedural data-flow tester. Technical report, The Ohio State University, Aug. 1997.

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

[8] M. Harrold and M. Soffa. An incremental approach to unit testing during maintenance. In *Proc. of the Conf. on Softw. Maint.*, pages 362–367, Oct. 1988.

[9] J. Hartmann and D. Robson. Techniques for selective revalidation. *IEEE Software*, 16(1):31–38, Jan. 1990.

[10] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. of the 16th Int'l. Conf. on Softw. Eng.*, pages 191–200, May 1994.

[11] J. Laski and W. Szermer. Identification of program modifications and its applications in software maintenance. In *Proc. of the Conf. on Softw. Maint.*, pages 282–290, Nov. 1992.

[12] H. Leung and L. White. A study of integration testing and software regression at the integration level. In *Proc. of the Conf. on Softw. Maint.*, pages 290–300, Nov. 1990.

[13] H. Leung and L. White. A cost model to compare regression test strategies. In *Proc. of the Conf. on Softw. Maint.*, pages 201–208, Oct. 1991.

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

[15] T. Ostrand and E. Weyuker. Using dataflow analysis for regression testing. In *Sixth Annual Pacific Northwest Softw. Qual. Conf.*, pages 233–247, Sept. 1988.

[16] D. Rosenblum and G. Rothermel. A comparative study of regression test selection techniques. In *Proc. of the 2nd Int'l. Workshop on Empir. Studies of Softw. Maint.*, Oct. 1997.

[17] D. Rosenblum and E. J. Weyuker. Lessons learned from a regression testing case study. *Empir. Softw. Eng. Journal*, 2(2), 1997.

[18] D. Rosenblum and E. J. Weyuker. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Transactions on Softw. Eng.*, 23(3):146–156, Mar. 1997.

[19] G. Rothermel. Efficient, effective regression testing using safe test selection techniques. Technical Report 96-101, Clemson University, Jan. 1996.

[20] G. Rothermel and M. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Softw. Eng.*, 22(8):529–551, Aug. 1996.

[21] G. Rothermel and M. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Softw. Eng. and Methodology*, 6(2):173–210, Apr. 1997.

[22] A. B. Taha, S. M. Thebaut, and S. S. Liu. An approach to software fault localization and revalidation based on incremental data flow analysis. In *Proc. of the 13th Annual Intl. Comp. Softw. and Appl. Conf.*, pages 527–534, Sept. 1989.