An Empirical Study of Regression Test Application Frequency

Jung-Min Kim **Adam Porter** **Gregg Rothermel**

ABSTRACT

Regression testing is an expensive maintenance process used to revalidate modified software. Regression test selection (RTS) techniques attempt to reduce the cost of regression testing by selecting and running a subset of an existing test suite. Many RTS techniques have been proposed in the research literature, and studies have shown that they can produce savings. Other studies have shown that the cost-effectiveness of RTS techniques can vary widely with various characteristics of the workloads (programs, versions, and test suites) to which they are applied. It seems plausible, however, that another set of factors impacting the cost-effectiveness of RTS techniques involves the *process* by which they are applied. In particular, issues such as the frequency with which regression testing is done have a strong effect on the behavior of RTS techniques. Therefore, in earlier work an experiment was conducted to assess the effects of test application frequency on the cost-effectiveness of RTS techniques. The results exposed essential tradeoffs that should be considered when using these techniques over a series of software releases. This work, however, was limited by several threats to external validity; in particular, the subject programs utilized were relatively small. Therefore, in this work, the previous experiment has been replicated on a large, multi-version program. This second experiment largely confirms the initial findings of the first study. In particular, results indicate that the cost of using safe RTS techniques was strongly and negatively affected by testing interval; that is, as the number of changes made to the program since the previous testing session increased, the number of test cases selected rose rapidly. Conversely, results show that the effectiveness of minimization RTS techniques was strongly and positively affected; that is, as the number of changes increased, so did the effectiveness of the test suites selected by minimization.

Keywords: regression testing, regression test selection, empirical studies

1 INTRODUCTION

After modifying software, developers typically want to know that existing system functionality has not been adversely affected. To obtain such knowledge, developers often perform *regression testing*. The simplest regression testing strategy is to rerun all existing test cases. This strategy is easy to implement, but can be unnecessarily expensive, especially when changes affect only a small part of the system.

Consequently, an alternative approach, regression test selection (RTS), has been extensively investigated [e.g., 1, 2,5, 6,10, 14]. With this approach only a subset of the test cases contained in a test suite are selected and rerun. Reducing the number of test cases rerun reduces regression testing costs, but may also cause fault-revealing test cases to be omitted. Since, in general, optimal test selection (i.e., selecting exactly the fault-revealing test cases) is impossible [13], the cost-benefit tradeoffs of RTS techniques are a central concern of regression testing research and practice.

A common way to empirically study this problem has been to find or create base and modified versions of a system and accompanying test suites. Next, one or more RTS techniques are run and the size and effectiveness of the selected test suites are compared to the size and effectiveness of the original test suite [e.g., 4, 11, 12, 16, 17].

Empirical studies of this sort have revealed several cost-effectiveness tradeoffs between RTS techniques, but they have also revealed that the performance of RTS techniques can vary widely with characteristics of programs, modifications, and test suites [16]. Recent studies [18, 22] have thus attempted to empirically evaluate some of these sources of differences.

One limitation of all this prior empirical work is that it fails to account for differences in testing processes, which may affect the cost-effectiveness of RTS techniques. In particular, previous studies of RTS techniques have all modeled regression testing as a one-time activity. In practice, however, regression testing is often a *continuous* process. For example, software releases often require many changes to a system with regression testing sessions interspersed between various numbers of changes rather than performed just once prior to product release. Similarly, many companies integrate system components and then regression test software changes on a monthly, weekly, or even daily basis.

One can hypothesize that the amount of change made between regression testing sessions strongly affects the costs and benefits of different RTS techniques. That is, it is possible that some RTS techniques will perform less cost-effectively as the amount of changes made between regression testing sessions grows. This is because they will select increasingly larger test suites and because these suites will become increasingly less cost-effective at finding faults. If this hypothesis is correct, testing practitioners may be able to better manage and coordinate their integration and regression testing processes by altering them to accommodate the effects of change size; this in turn may result in savings in time and money.

To test this hypothesis, in earlier work [20] an experiment was conducted to assess the effects of test application frequency on the costs and benefits of RTS techniques. The results exposed essential tradeoffs that should be considered when using these techniques over a series of software releases. This work, however, was limited by several threats to external validity; in particular, the subject programs utilized were relatively small. Therefore, in this work, the previous experiment is replicated on a large, multi-version program. This second experiment largely confirms the findings of the first. In particular, the cost of using safe RTS techniques was strongly and negatively affected by testing interval; that is, as the number of changes made to the program since the previous testing session increased, the number of test cases selected rose rapidly. Conversely, the effectiveness of minimization RTS techniques was strongly and positively affected; that is, as the number of changes increased, so did the effectiveness of the test suites selected by minimization.

The remainder of this paper reviews the relevant background material and literature, and then presents the design and analysis of both the initial experiments and the replication of those experiments. Finally, the results of the two experiments are compared, and then conclusions and future directions for research are presented.

2 BACKGROUND 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 test cases 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 test cases 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''.

Each of these steps involves important problems. However, this work concerns only step 1 - the *regression test selection problem.*

2.2 Regression Test Selection Techniques

Several RTS techniques have been investigated in the research literature (see [13]). Here several classes of techniques are briefly described, and a representative example of each is presented.

Retest-All. This approach reruns all test cases in T. It may be used when test effectiveness is the utmost priority with little regard for cost.

Random/Ad-Hoc. Testers often select test cases randomly or rely on their prior knowledge or experience. One such technique is to randomly select a percentage of test cases from T.

Minimization. These approaches (e.g., [3, 6]) aim to select a minimal set of test cases from *T* that covers all modified or affected elements of P'. One such technique randomly selects test cases from *T* until every statement added to or modified in creating P' is exercised by at least one test case.

Safe. These approaches (e.g., [2,14]) select, under certain conditions, every test case in *T* that covers changed program entities in P'. One such technique [14] selects every test case in *T* that exercises at least one statement that was added to or modified in creating P', or that has been deleted from *P*.

2.3 Cost and Benefit Models

Leung and White [8] present a model of the costs and benefits of RTS strategies. Costs are divided into two types: *direct* and *indirect*. Indirect costs include management overhead, database maintenance, and tool development. Direct costs include costs of test selection, test execution, and results analysis. Savings are simply the costs avoided by not running unselected test cases.

Let T' be the subset of T selected by a certain RTS technique M for program P, and let $|T'|$ denote the cardinality of T'. Let s be the average cost per test case of applying M to P to select T', and let r be the average cost per test case of running P on a test case in T and checking its result. Leung and White argue that for RTS to be cost-effective the inequality: $s|T'| < r(|T| - |T'|)$ must hold. That is, the analysis required to select T' should cost less than the cost of running the unselected test cases, $T - T'$.

One limitation of this model is that it overlooks the cost of undetected faults. Since a primary purpose of testing is to detect faults, it is important to understand whether, and to what extent, test selection reduces fault detection effectiveness. To address this limitation, Malishevsky et al. [21] extend Leung and White's to factor in benefits related to fault detection effectiveness.

2.4 Previous Empirical Studies

Initially, cost-effectiveness, as defined by Leung and White, was the central focus of regression test selection studies.

Rosenblum and Weyuker [12] applied the technique **TestTube** to 31 versions of the KornShell and its test suites. For 80% of the versions, their method selected all existing test cases. They note that the test suite is relatively small (16 test cases), and that many of the test cases exercise all the components of the system.

Rothermel and Harrold [14] conducted a similar study with their technique, **DejaVu**, using several 100-500 line programs and a larger (50 KLOC) program. The savings averaged 45% for small and medium sized programs, and 95% for the larger program.

These two studies seem to indicate that in some cases, regression test selection can be cost-effective. Later studies, therefore, began to compare different methods, and also to examine fault-detection effectiveness.

Bible et al. [11] compared the performance of **TestTube** and **DejaVu** in terms of test selection and fault-detection effectiveness. The two techniques often performed similarly, but in some cases **DejaVu** substantially outperformed **TestTube**.

Graves et al. [4] examined the relative costs and benefits of several RTS techniques. They examined five techniques: *minimization, safe, dataflow, random, and retest-all,* focusing on their abilities to reduce test suite size and to detect faults. The study revealed several results. First, the safe technique detected all faults while on the average selecting 68% of the test cases; however, it sometimes selected all test cases. Second, the safe and dataflow techniques performed nearly identically; they typically detected the same faults while selecting the same numbers of test cases. Third, on average, random test suites were nearly as effective as those selected by the safe technique. Finally, minimization yielded the smallest and the least effective test suites; for example, small random test suites (with 5 or so test cases) were equally effective at finding faults, but required no analysis.

Data from the studies described above revealed that several factors involving workload (programs, versions, and tests suites) could affect RTS technique performance. To begin to examine these factors, Elbaum et al. [22] studied the effects of change size on the effectiveness of RTS techniques. This study showed that change distribution has a greater effect on technique performance than change size; however, change attributes and test case interaction patterns interact in such effects.

Subsequently, Rothermel et al. [18] investigated the impact of test suite granularity (a measure of the way in which test cases are grouped into test suites) on RTS techniques. They considered four techniques: safe, modified-non-core-entity (like safe techniques, but ignoring changes in core functions, thus trading some safety for efficiency), minimization, and retest-all. They measured the effects of test suite granularity on regression test execution time and fault detection effectiveness, across these techniques, on several versions each of two large software systems. Results indicated that fine granularity suites (utilizing relatively large numbers of small tests) are more supportive of test selection than coarse granularity suites, but that granularity effects are also associated with fault-detection effectiveness effects.

2.5 Open Questions

Most previous studies of regression test selection have focused on the choice of RTS technique. This is obviously an important issue. However, this ignores another equally important issue – the application policy. That is, what are the conditions that trigger regression testing: periodic execution (daily, weekly, or monthly), rule-based execution (after all changes, after changing critical components, or at final release), or something else?

It seems likely that application policy is important because it may greatly affect the practical costs and benefits of regression test selection. Therefore, this paper investigates application policies, by considering how the amount of change made to a system between regression testing sessions affects the costs and benefits of different RTS techniques. In particular, for different RTS techniques the goal is to determine the following:

- How, as the amount of change between two software versions increases, do test suite reduction and fault detection effectiveness change?

- As amount of change increases, what tradeoffs exist between test suite reduction and fault detection effectiveness?
- As amount of change increases, when is one RTS technique more cost-effective than another?

The next three sections first describe an initial experiment investigating these questions; then, the replicated experiment performed for this work is presented, and finally the two studies are compared. The complete discussion of the initial experiment appears in Kim et al. [20].

3 INITIAL EXPERIMENT

3.1 High-Level Hypotheses

Two high-level hypotheses are addressed, addressing cost and effectiveness factors in turn:

> **H1:** (cost) test selection ratios change as the number of modifications made between regression testing sessions changes;
>
> **H2:** (effectiveness) fault detection ratios change as the number of modifications made between regression testing sessions changes.

3.2 Measures

To investigate these hypotheses it was necessary to measure the costs and benefits of each RTS technique. To facilitate this, two models were constructed: one for calculating savings in terms of test suite size reduction, and another for calculating costs in terms of fault detection effectiveness. Attention was restricted to these costs and benefits, but there are many other costs and benefits these models do not capture (some of these are outlined in Section 3.4.1).

Measuring Savings. Reducing test suite size provides savings because it allows testers to run fewer test cases, examine fewer test results, and manage less test data. These savings are proportional to the reduction in test suite size. Thus, in this work, savings are measured in terms test selection ratio, as given by $|T'|/|T|$.

This approach makes several simplifying assumptions. It assumes that the cost of all test cases is uniform and all the constituent costs can be expressed in equivalent units (e.g., no differentiation is made between CPU time and human effort). It also does not measure the savings that may result from applying analyses, done for early testing sessions, to later testing sessions.

Measuring Costs. Two types of costs are considered. The first comes from the analysis needed to select test cases. The second may occur when the selected test cases do not detect faults that could have been detected by the original test set. The cost model used for this experiment focuses on the latter cost.

To determine whether regression test selection reduces fault detection effectiveness, it is necessary to measure which test cases reveal which faults in P'. However, there is no simple way to determine this because when a test case fails on a program that contains multiple faults it is not always obvious exactly which fault(s) caused the failure. Thus, three estimators were considered.

Estimator 1 - On a per-test-suite basis. One way to measure test effectiveness is to classify the selected test suite into one of three cases: (1) no test case in T detects faults, and thus, no test case in T' detects faults; (2) some test cases in both T and T' detect faults; or (3) some test cases in T detect faults, but no test case in T' detects faults. Cases 1 and 2 indicate test selection that does not reduce fault detection, and case 3 captures the situation in which test selection compromises fault detection.

This method is imprecise because it treats all faults in P' as a single fault. The main advantage of this method is that it is inexpensive to implement.

Estimator 2 - On a per-test-case basis. Another approach is to identify those test cases in T that detect faults in P' but that are not included in T'. The number of test cases in T that detect faults in P' then normalizes this quantity.

This approach is also imprecise because it assumes that every fault revealing test case reveals a different fault. When multiple test cases reveal the same fault, duplicate test cases could be discarded without sacrificing fault detection effectiveness. This measure penalizes such a decision.

Estimator 3 - On a per-fault basis. This approach tries to identify all test cases that might "theoretically" reveal each fault. A test case t that detects a fault f must satisfy three conditions: (1) t must traverse the program point containing f in P', (2) immediately after t traverses the program point containing f in P', program state must be perturbed (3) the final program state of P' for test case t must be different from that of P run on test case t [19].

Using this information, one can determine which faults may be detected by each test case. This method is the most precise, but because it requires hand-instrumentation it is also the most expensive.

For this study, this third approach was selected and implemented as follows. To detect the first condition, every program modification point in P' was instrumented to determine whether t traversed the program point containing f. To detect the second condition the program was further instrumented, creating two blocks – one with the faulty code and one without. The state of all global and in-scope local variables was captured immediately before the change, then both blocks were executed, and then their states upon exit were compared. If these states differed then it was concluded that t perturbed the program state, thus satisfying the second condition. To detect the third condition, the output of P and P' was compared to identify whether they produced different outputs for test case *t*.

After this analysis had been performed for each test case in *T*, the number of faults for which there existed at least one fault-revealing test case in *T* was counted. This number was called NF_{det}. Next, *T′* was examined, and again the number of faults for which there existed at least one fault-revealing test case was counted. This number was called NF_{det}'. Finally, the total number of faults in *P′* was designated as *NF*.

These numbers were used to calculate two measures of effectiveness. One is *relative effectiveness,* defined as NF_{det}'/NF_{det} and the other is *absolute effectiveness,* defined as NF_{det}'/NF.

3.3 Experiment Instrumentation

Programs, Tests and Faults. For this experiment, eight C programs, with a number of modified versions and test suites for each program, were utilized. The subjects come from two sources. One is a group of seven C programs collected and constructed initially by Hutchins et al. [7] for use in experiments with dataflow- and control-flow-based test adequacy criteria. The other, Space, is an interpreter for an array definition language (ADL) used within a large aerospace application. Table 1 describes the subjects, showing the number of functions, lines of code, distinct versions, test pool size, and the size of the average test suite. These and other aspects of the data are described in the following paragraphs.

Siemens Programs: Seven of the subject programs come from a previous experiment by Hutchins et al. [7]. These programs are written in C, and range in size from 7 to 21 functions and from 138 to 516 lines of code.

Program	Functions	LOC	# 1st-order versions	Tot. Size	Avg Suite Size
replace	21	516	12	5542	398
printtokens2	18	402	7	4130	318
printtokens	19	483	9	4115	389
schedule	18	299	7	2650	225
schedule2	16	297	8	2710	234
tcas	9	138	12	1608	83
totinfo	7	346	12	1054	199
space	136	6218	10	13585	4361

Table 1: Experiment Subjects

For each of these programs Hutchins et al. created a pool of black-box test cases [7] using the *category partition method* and Siemens Test Specification Language tool [8]. They then augmented this set with manually created white-box test cases to ensure that each exercisable statement, edge, and definition-use pair in the base program or its control flow graph was exercised by at least 30 test cases.

Hutchins et al. also created *faulty* versions of each program by modifying code in the base version; in most cases they modified a single line of code, and in a few cases they modified between 2 and 5 lines of code. Their goal was to introduce faults that were as "realistic" as possible, based on their experience with real programs.

Ten people performed the fault seeding, working "mostly without knowledge of each other's work" [7, p. 196]. To obtain meaningful results, the researchers retained only faults that were detectable by at least 3 and at most 350 test cases in the associated test pool.

Space Program: The Space system, written in C, is an interpreter for an array definition language (ADL). The program reads a file that contains ADL statements, and checks the contents of the file for adherence to the ADL grammar, and to specific consistency rules. If the ADL file is correct, Space outputs an array data file containing a list of array elements, positions, and excitations; otherwise the program outputs error messages.

Space has 38 versions, each containing a single fault that was discovered either during the program's development or later by the authors of this study.

The test pool was constructed in two phases. First a pool of 10,000 randomly generated test cases, created by Vokolos and Frankl [17], was obtained. Then, new test cases were added until every dynamically executable edge in the program's control flow graph was exercised by at least 30 test cases (excluding those edges that can be exercised only by the occurrence of malloc faults). This process yielded a test pool of 13,585 test cases.

Versions. In this experiment, programs with varying numbers of modifications were needed. These were generated in the following way. Each subject program initially consisted of a *correct* base version and a number of modified versions, each containing exactly one *fault*: these are called 1st-order versions. These versions were selected because the faults they contain are "*mutually independent*." That is, any number of these faults can be merged into the base program simultaneously. For example, if fault f_1 is caused by changing a single line and fault f_2 is caused by deleting the same line, then these modifications interfere with each other. Table 1 shows the number of 1st-order versions for each subject program, ranging from 7 to 12.

Next, higher-order versions were created by combining appropriate 1st-order versions. For example, to create an nth-order version, *n* unique 1st-order versions were combined. This was done for every subject program until all possible *k*th-order versions had been created, where *k* was the minimum of 10 or the number of 1st-order versions available for that program. As an example, the `tcas` program had 12 1st-order versions. Therefore, 12 1st-order versions, 66 2nd-order versions, and so on up to 66 10th-order versions were constructed, for a total of 4082 versions. An exception was made, however, for the Space program. Since its test suites are much larger than those of the Siemens programs - they take 10-100 times longer to run – the number of 1st-order versions for the Space program was limited to 10.

In this way the regression testing of systems in which varying amounts of modifications have been made since the previous regression testing session was modeled.

Test Suites. The test pools for the subject programs were used to obtain two types of test suites for each program: edge-coverage-adequate test suites and random test suites (non-coverage-based). To create edge-coverage-adequate test suites, the test pool for the base program, and test coverage information gathered from the test cases, was used to generate 1000 edge-coverage-adequate test suites for each base program.

1000 randomly-selected test suites were also generated for each base program. To generate the kth random test suite T for base program P ($1 \leq k \leq 1000$), n, the number of test cases in the kth edge-coverage-adequate test suite for P, was determined. Next, test cases were chosen at random from the test pool for P and added to T until it contained n test cases. This process yielded random test suites of the same size as the edge-coverage-adequate suites.

Regression Test Selection Tools. To perform the experiments, implementations or simulations of RTS techniques were needed. For *safe* techniques an implementation of Rothermel and Harrold's **DejaVu** tool [14, 15] was used, and another safe technique, **TestTube** [2], was simulated. For *minimization*, a tool that selects a minimal test suite T' was selected, such that T' had at least one test case that covers every node in the control flow graph for P that was changed between P and P'. As a random technique a tool called *random(n)*, that randomly selects $n\%$ of the test cases from the suite, was created. *Retest-all* did not require any tools (because all the test cases are selected).

3.4 Experiment Design

The experiment manipulated four independent **variables**:

1. The subject program (there are 8 programs, each with a variety of modified versions).

2. The RTS technique (one of **DejaVu**, **TestTube**, minimization, retest-all, random(25), random(50), random(75)).

3. Test suite composition (edge-coverage-adequate or random).

4. Test interval (from base to modified program, from 1 to 10 changes can be made).

For each combination of program, test interval and technique, 100 edge-coverage-adequate test suites and 100 random test suites were used. On each test run, with base program P, modified version P', technique M, and test suite T, the following measurements were collected:

1. The proportion of test cases selected in T' to test cases in the original test suite T.

2. The number of faults revealed by T and T'.

From these data points, two dependent variables were calculated:

1. Average selected test suite size.

2. Average fault detection effectiveness.

The experiment used a full-factorial design with 100 repeated measures. That is, for each subject program, test interval and test suite composition criterion,100 test suites were selected from the test suite universe. For each test suite, each RTS technique was applied, and the size and fault detection effectiveness of the selected test suites was measured. In total, 20,004,600 test suites were executed and evaluated.

3.4.1 Threats to Validity

Threats to internal validity are influences that can affect the dependent variables without the researcher's knowledge. They can thus affect any supposition of a causal relationship between the independent and dependent variables. In this study, the greatest concern was that instrumentation effects could bias the results. Instrumentation effects may be caused by differences in the experimental instruments (in this case the test process inputs: the code to be tested, the locality of the program changes, the composition of the test suite, or the composition of the series of versions). One related issue is that all modifications to the subject programs were considered faults. In reality, some modifications will not result in faults. In this study two different criteria were used for composing test suites: edge-coverage-adequate and random. In order to reduce effects due to program versions, all possible combinations of versions were used. However, there was no attempt made to control for the structure of the subject programs, or for the locality of program changes. To limit problems related to this, the RTS techniques were run on each suite and each subject program.

Threats to external validity are conditions that limit one's ability to generalize the results of an experiment to industrial practice. One threat to external validity concerns the representativeness of the subject programs. The subject programs are of small and medium size, and larger programs may be subject to different cost-benefit tradeoffs. Also, the Siemens programs contain seeded faults although efforts were made to make them as realistic as possible. Another issue is that these faults are roughly the same "size". Therefore, a program with, say, ten faults has been changed more than a program with one fault. Also, the faults utilized are mutually independent whereas naturally-occurring faults may sometimes overlap or obscure one another. In short, industrial programs have much more complex error patterns. Another threat to external validity for this study is process representativeness. This arises when the testing process used is not representative of industrial ones. This may endanger the results of this study since the test suites utilized may be more or less comprehensive than those that could appear in practice. Also, this experiment mimics a corrective maintenance process where the specification is not changed, but there could be many other types of maintenance in which regression testing might be used. These threats can be addressed only through additional studies using a greater range of software artifacts. In fact, Section 4 contains a first attempt to replicate this experiment in order to address these concerns.

3.5 Data and Analysis

In this paper, extensive use is made of box plots (e.g., Figure 3) to represent data distributions. In these plots, a box represents each distribution. The box's width spans the central 50% of the data and its left and right ends mark the upper and lower quartiles. The bold dot within the box denotes the median. The dashed horizontal 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 considered "outliers". Typically, one box plot will present several boxes side by side to allow for visual comparison of the distributions. For example, Figure 3 shows the relative effectiveness of the retest-all technique grouped by testing interval. Therefore, each box labeled "i" represents only the data from test runs with testing interval i.

In many cases, multiple box plots are combined to allow grouping by a third variable. For example, Figure 1 is composed of 7 box plots, where each box plot contains the data for 1 specific RTS technique. Each individual box plot then shows the distribution of test selection size broken down by testing interval for just that one technique.

The data are analyzed in three steps. First, the ability of different RTS techniques to reduce test suite size and still detect faults as the testing interval grows is compared. Second, the effectiveness of the original test suite is examined as the testing interval grows. Finally, the cost-benefit tradeoffs of program-analysis-based (i.e., safe and minimization) and random techniques are compared, and the factors that may be responsible for the differences between them are discussed.

Note that this analysis does not rely on statistical null hypothesis testing. There are several reasons for this, including the facts that the number of versions for each subject program was quite different so only a small number of programs we represented at the highest testing intervals, and that the large number of sub-hypotheses investigated might chop up the data too finely. These and other factors led to concerns about violating certain assumptions underlying traditional statistical tests. Therefore, the analyses performed here are restricted to graphical techniques to visualize the data and descriptive statistics for all distributions of interest.

Size Reduction. Figure 1 shows the ability of each RTS technique to reduce test suite size by testing interval, conditioned on the technique itself. The random(*n*) methods selected *n%* of the test cases by construction and the retest-all method always selected all test cases. Therefore only program-analysis-based methods are considered here.

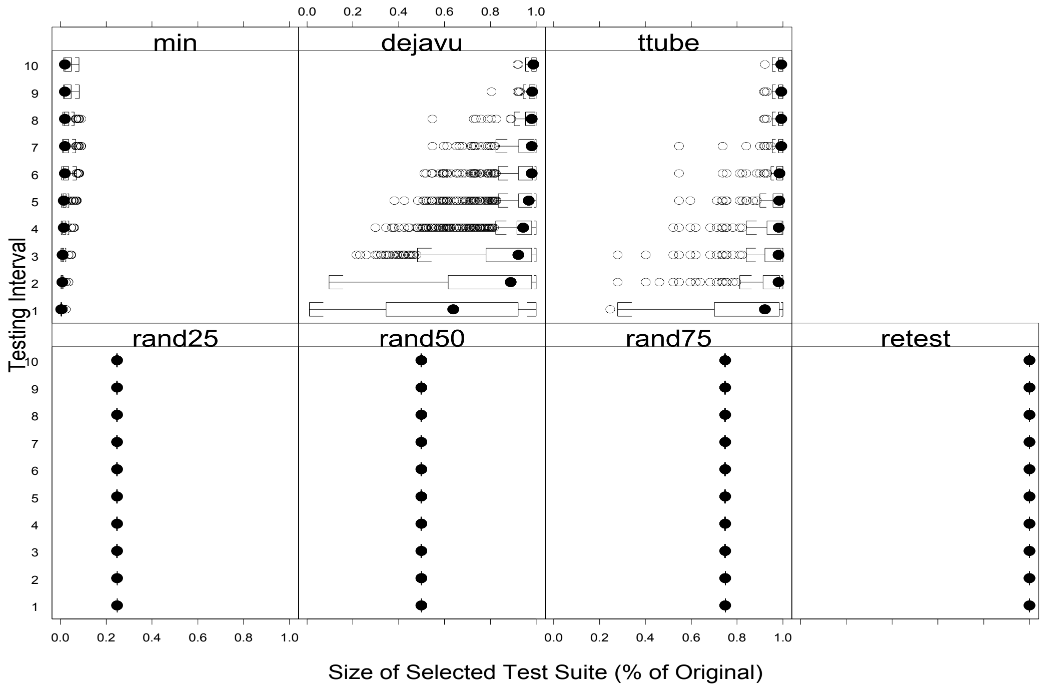

Figure 1: Test Selection Ratio by Testing Interval Conditioned on RTS Technique

First, observe that **DejaVu** selected a median of 66.5% of the test suite when the testing interval was 1 (as was found in earlier experiments [4]). However, the median ratio increased rapidly as testing interval increased. For example, **DejaVu's** median test selection across all programs and intervals was 97.4%. Inspection of the programs suggested that selection ratio was heavily dependent on the program, and the type and location of code changes. For example, **DejaVu's** selection ratios ranged from 2.2% to 98.1% for `printtokens` for testing interval 1.

Next, note that **TestTube** selected a median of 91.8% of the test suite when the testing interval was 1. As the testing interval increased, selection ratios increased to 99.3%.

Minimization selected a median of 0.4% of the test suite for interval 1 to a median of 2.0% for interval 10. This increase is less than one new test case for each added change. For example, for `totinfo`, a median of 4.2 test cases was selected when testing interval was 10 and 1 test case

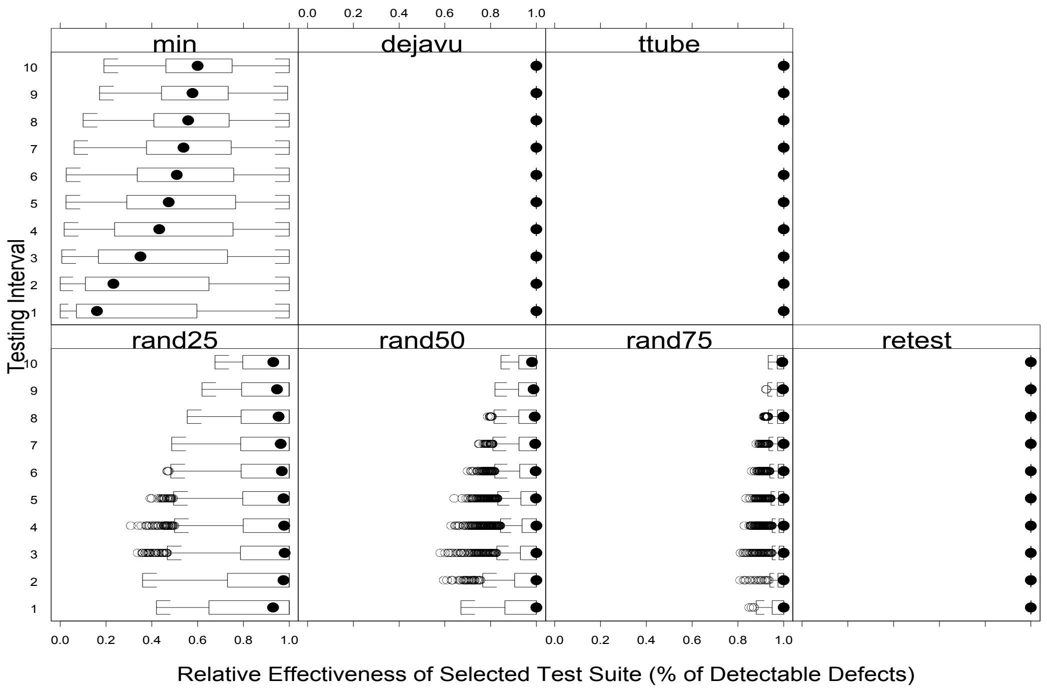

Figure 2: Relative Effectiveness by Testing Interval Conditioned on Test Selection

was selected when the testing interval was 1.

Relative Effectiveness. Figure 2 shows the relative effectiveness of selected test suites by testing interval, broken down by technique used.

Safe methods, by virtue of being "safe", guarantee that all "detectable" faults will be detected (given certain assumptions outlined in [2,14]). That is, their effectiveness is the same as that of retest-all. Therefore, this section concentrates on the random and minimization methods.

For random methods, the median relative effectiveness was always over 90% when the testing interval was 1. As testing interval increased, relative effectiveness also increased up to a point and then stayed essentially flat. For example, the median effectiveness of random(25) was 96.8%. Another thing to note is that, on average, random(50) and random(75) were nearly as effective as retest-all – with median relative effectiveness of 99.7% and 100%, respectively.

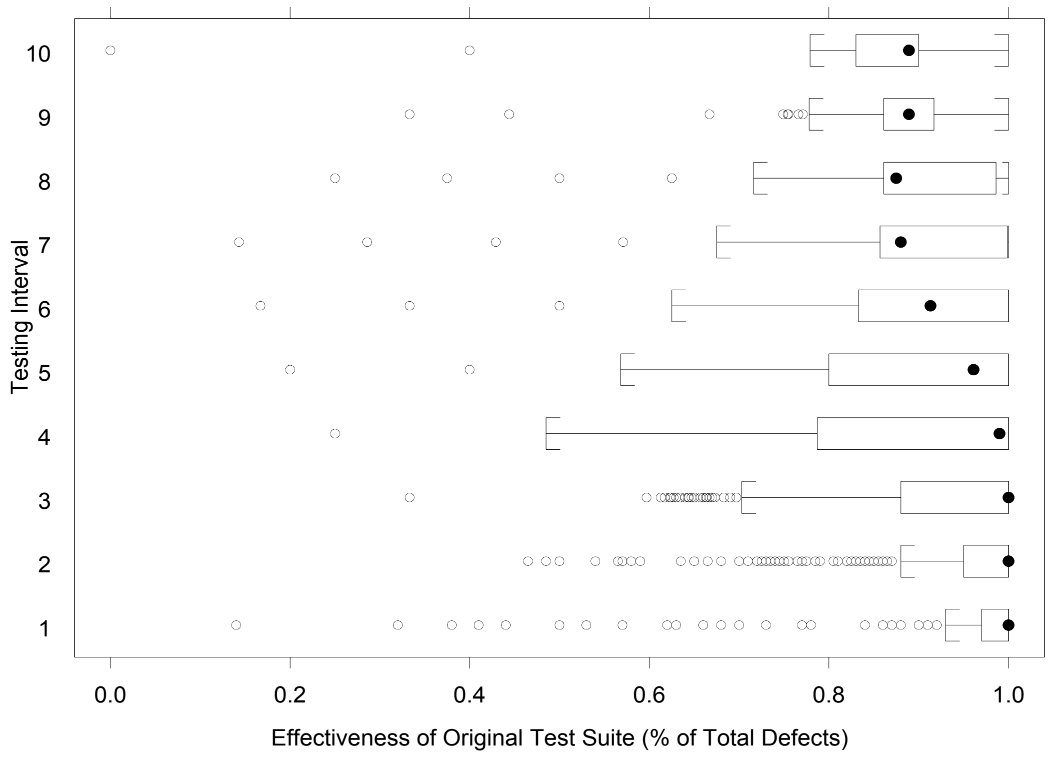

Figure 3: Absolute Effectiveness for Retest-All by Interval

Minimization had a median relative effectiveness of 16.0% at interval 1. However, effectiveness climbed rapidly as the interval increased – with a median relative effectiveness of 60.0% at interval 10.

Effectiveness of T. During this study, it was hypothesized that a test suite that revealed a fault *f* in *P′* when *f* was the only fault in *P′* might no longer reveal the same fault when *f* is mixed with other faults. One possible reason is that failures can be "hidden" by interacting faults. Another may be that some faults may change a program's control flow, causing other faults to go unexecuted. If this happens, then increasing the testing interval may negatively affect not only the fault-detection effectiveness of RTS techniques, but of retest-all as well.

This hypothesis was evaluated by running the original suite *T* on instrumented *P′*. The number of faults in *P′* that were "detectable" by *T* was then counted (see Section 3.2 for the conditions under which we consider a fault to be detectable).

Figure 3 shows the percentage of detectable faults broken down by testing interval. By construction, it is necessarily the case that the fault in each 1st-order version was detectable by some test cases in the test "pool" associated with that program. However, there were some test suites that did *not* detect particular faults.

As testing interval increased, the percentage of detectable faults dropped steadily, with a median of 88.9% when the testing interval was 10. However, since most of the higher-order (after 8th-order) versions come from three subject programs (`tcas`, `totinfo`, and `replace`), the behavior after interval eight needs to be interpreted with caution. Nevertheless, the data are consistent with the hypothesis that fault detection effectiveness decreased as the number of faults increases. Therefore, the absolute effectiveness of selected test suites was also investigated.

Absolute Effectiveness. Figure 4 shows the absolute effectiveness of selected test suites by test interval, conditioned on technique.

Safe techniques always had a relative effectiveness of 100%. However, their absolute effectiveness dropped as interval increased. Random techniques showed a similar pattern. In contrast, the median absolute effectiveness of minimization increased as the testing interval grew.

3.6 Cost-Benefit Tradeoffs

When both the costs and the benefits of the different RTS techniques are considered, several interesting patterns emerge for the Siemens/Space programs.

Overall, **DejaVu** selected 97.4% of the test cases while showing 91.8% absolute effectiveness,

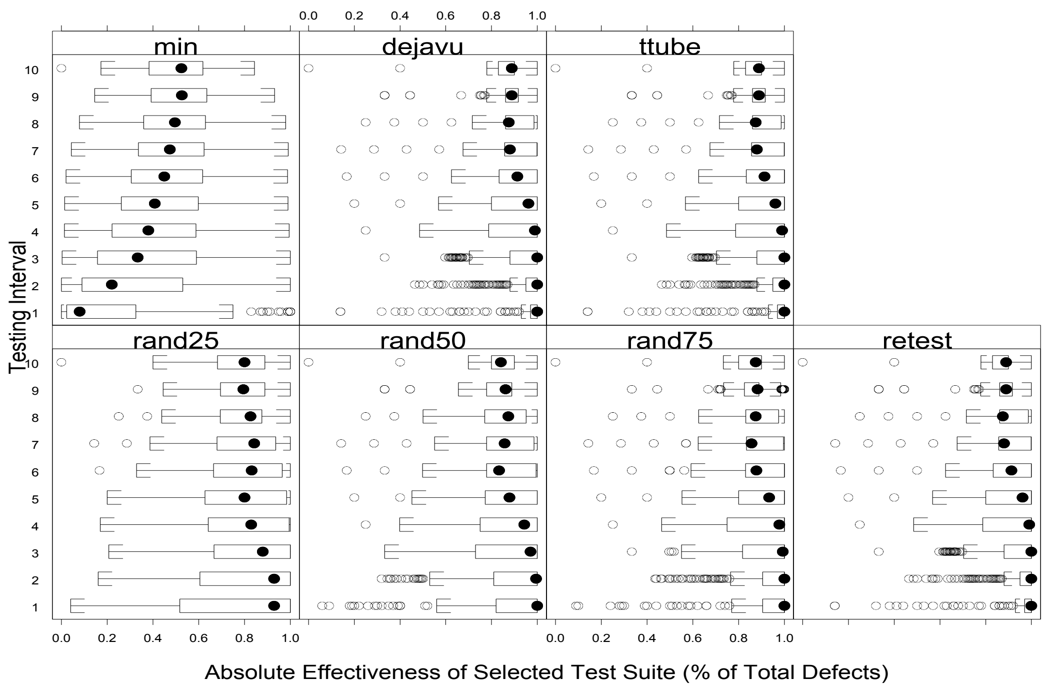

Figure 4: Absolute Effectiveness by Testing Interval Conditioned on RTS Technique

but it was most cost-effective when the testing interval was small (66.5% of test cases were selected). **TestTube** also performed best under these circumstances although the difference was not as great. These techniques may also be cost-effective when the cost of missing faults is very high or when the cost of running test cases is very high. For example, for some safety-critical systems the cost of missing a fault may be so high that non-safe RTS techniques cannot be used.

Across all observations, minimization selected 1.5% of the test cases and showed 44.2% absolute effectiveness. It was most cost-effective when the interval was large – at interval 10 it selected 2.0% of the test cases while showing 52.4% absolute effectiveness. Minimization might therefore be recommended when the cost of running test cases is very high and the cost of missing faults is not too high.

All random techniques had high effectiveness at low testing intervals and increasing effectiveness as testing interval increased. As testing interval increased, differences between different random methods decreased. Thus, if the cost of missing faults is not too high, then a small random technique, random(25), would be the most cost-effective when the testing interval is high.

4 EXPERIMENT 2

In an attempt to address some of the threats to validity found in our initial experiment, the experiment was replicated using a larger, real-world software system.

4.1 Hypothesis

It was hypothesized that the behaviors seen in Experiment 1 would repeat themselves in larger software artifacts. That is, test selection ratios and test effectiveness ratios change as the number of modifications made between regression testing sessions changes. Therefore, a large, real-world program called DNAM (DNA Mapping) was acquired, and Experiment 1 was replicated on this new program to see if this hypothesis would remain true.

4.2 Measures

Measuring Savings. The savings model already defined in Section 3.2 was used ($|T'|/|T|$).

Measuring Costs. DNAM is different from the Siemens/Space programs in many respects. One difference is that DNAM has successive versions, not a base version with faulty derivatives. In addition, for DNAM, the changes between successive versions are not always faults, but include successful updates to fix bugs, enhance performance, and add new functionality. Therefore, this experiment focused on modification points (MPs) rather than faults.

Cost model on a per-MP basis: The cost model is similar, but not identical, to the one used in Experiment 1. One important difference stems from focusing on modification revealing test cases rather than fault revealing ones. Here, the approach used is to identify all test cases that might “theoretically” reveal each MP. A test case t that detects an MP m must satisfy three conditions (similar to those needed to reveal a true fault [23]): (1) t must traverse the program point containing m in P', (2) immediately after t traverses the program point containing m in P', program state must be perturbed, and (3) the final program state of P' for test case t must be different from that of P run on test case t.

Using this information it can be determined which MPs can be detected by each test case. The first and second conditions are determined just as in Experiment 1. To detect the third condition, however, some manual inspection was required. For the Siemens/Space programs the final program states of P and P' can be assessed for differences by comparing their outputs. For DNAM, however, it is possible that the outputs of P' and P may be different, while their final program states are not; for example, because of changes to formatting code, changes to menu text, or when timestamps are embedded in program output. Consequently, it was necessary to manually compare the outputs of successive versions to decide whether their output differences stemmed from functional changes.

Another difference between DNAM and earlier subjects is that DNAM’s test pool does not cover every exercisable edge and thus every potential modification in the program. In fact only 264 MPs of the 1457 total MPs (18%) are traversed by at least one test case in the test pool. This means that at least 82% of the time neither T nor T' detect the MP. These unexercised MPs would significantly bias the cost model used in Experiment 1. Therefore, it was necessary to focus only on “exercisable” MPs.

After the analysis for each test case in T, the number of MPs for which there exists at least one revealing test case in T is counted. This number is called NMP_{det}. Next, T' is examined again, and the number of MPs for which there exists at least one revealing test case is counted. This number is NMP_{det}'. Finally, the total number of exercisable MPs in P' is designated as NMP. These numbers are used to calculate *relative effectiveness,* defined as NMP_{det}'/NMP_{det}.

4.3 Experimental Instrumentation

DNAM is a DNA mapping program developed in C by a research group from Washington University in St. Louis. It consists of 216 C files and 216 header files with 8414 functions resulting in 205 KLOC (non-comment, non-blank) with 232 test cases in the test pool. DNAM functions as follows. In a biological laboratory, pieces of DNA called genomes are broken up into overlapping pieces called clones. One specific type of clone is called a restriction fragment. DNAM reads a command file and a file describing restriction fragments as input and produces maps explaining how the clones are related to one another as output. DNAM's input files are obtained from the biology laboratory.

From an experimenter's point of view, DNAM is a mixed blessing. Its greatest advantage is that it is not a "toy" program. Rather, it is a large system with complex behaviors and change patterns. Nevertheless, it has several limitations. First, the configuration management process used during its development appears to have resulted in check-ins roughly once per week, so its fine-grained change history has been lost. Second, no control can be had over the system's development process, so change size is not uniformly distributed. This limits the reasoning that can be made about the effect of change size. Third, as with many software systems, DNAM's test suite is seriously inadequate, leaving large portions of the system untested. This problem is compounded by the fact that the test inputs must correspond to actual pieces of DNA, making it impossible for us to easily obtain more test data. It was necessary to work within these limitations.

Versions. The DNAM source code resides in a Revision Control System (RCS) repository. The repository records run from June 1997 through February 1998. While setting up this experiment, daily source snapshots were retrieved using the RCS checkout utility's date/time option. It was discovered that, out of a possible 246 daily snapshots, only 40 versions contained non-cosmetic changes. Of these, the amount of change between successive versions ranged from very small (just a few lines in one file) to quite large (up to 15K lines spread across 16 C files). To smooth out these size differences, "intermediate versions" were generated from several of the greatly changed versions.

In creating the intermediate versions, great care was taken to generate incrementally changing versions. First the changes between two successive versions were reviewed, in an attempt to find a series of logically related partial deltas. For example, in one case a new function had been created. Therefore, the C file containing the new function and all changes to C files from which the newly created version was called formed a logically related partial delta. Next, an attempt was made to build the system with only the partial delta. It successfully linked and ran. This process was then repeated, identifying a new partial delta from the remainder of the original delta (previous delta minus the partial delta). These steps continued until every change in the original delta was contained in one of the partial deltas. Obviously, this job couldn't be automated, but through this trial-and-error methodology, 102 successive versions were created.

Test Cases, Test Pools, and Test Suites. Initially DNAM had 232 test cases in its test pool. Each test case is based on an input clone file provided from the biology laboratory, and a command file. These test cases do not cover each statement; in fact, 64 of 102 versions do not have any MP-revealing test cases. There is also a significant imbalance between the number of test cases exercising some functions and the number of modifications applied to them.

DNAM consists of five major libraries (i.e., libmstrat, libmap, libstruct, libadt, and libutil), and

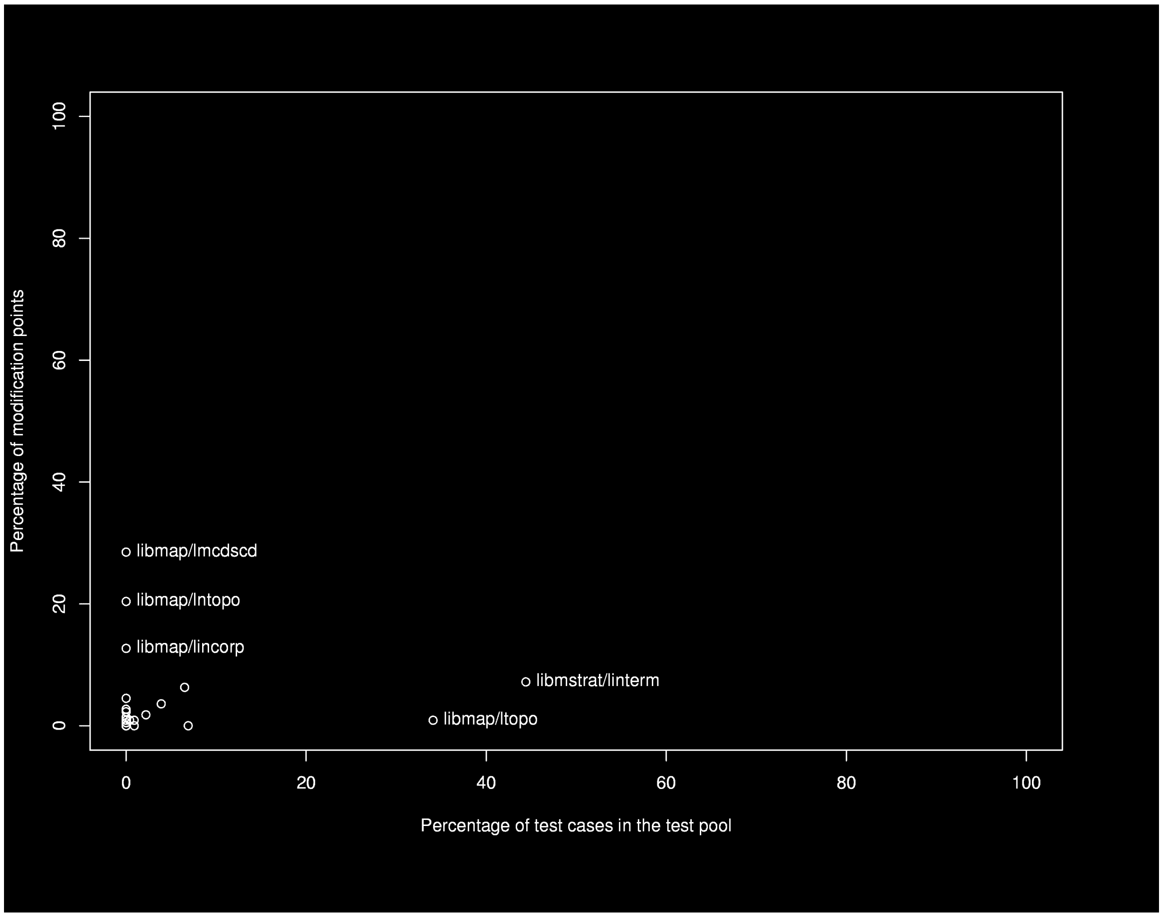

Figure 5: Distribution of Modifications and Test Cases for Each Library

---

each library is composed of 3 to 11 small libraries. Figure 5 shows the distribution of modifications applied to each small library and the percentage of test cases in the test pool covering each small library. For example, almost half the test cases in the test pool (103 of 232) test the functions in libmstrat/linterm, while the library itself is modified in only 16 versions (about 7%). Another extreme example is that three libraries (i.e., libmap/lntopo, libmap/lmcdscd, and libmap/lincorp) were modified in 62% of the versions, but there are no test cases that exercise the functions in those libraries.

Although adding new test cases would have been desirable, this turned out to be impossible. Test cases for this system are based on actual DNA sequences, making it impossible for the experimenters to generate them by themselves. Moreover, the developers who built DNAM are no longer maintaining it and are unable to provide additional test cases.

The test cases in the test pool test sixty-four different system functions. To create test suites, each test case was categorized by the functions it tests. Then, one test case was selected from each category and added to the test suite. Finally, the test suite was augmented with randomly selected test cases. The selected subset of test cases was called function-coverage-adequate. 100 function-coverage-adequate test suites and 100 random test suites (whose size was the same as their corresponding function-coverage-adequate suites) were created. Each function-coverage-adequate test suite covered each of the 64 functions at least once. The average size of the test suites was 150.

Regression Test Selection. For DNAM, it was necessary to possess simulations of safe and minimization RTS techniques. To do this the code was hand-simulated at the entry to each modification point to report the set of test cases that reached that point. For a safe technique, then all test cases through each point were selected. For minimization, one test case through each point was selected. When all selection was complete, duplicate test cases were eliminated. This use of simulation affects tool analysis time, not test selection results, and thus does not impact results. The random(n) and retest-all techniques were implemented as in Experiment 1. To speed up the experiment, the TestTube method was omitted for this study.

4.4 Experimental Design

The experiment manipulated several independent **variables**:

1. The subject program version (there are 32 usable versions).

2. The RTS technique (one of **DejaVu**, minimization, retest-all, random(25), random(50), random(75)).

3. Test suite composition (function-coverage-adequate or random).

Unlike in Experiment 1 there is no testing interval variable. This is because the testing interval is dictated by the actual data, not set artificially as it was in Experiment 1. For each technique, 100 function-coverage-adequate test suites and 100 random test suites were run. For each testing session, with base program *P*, its modified successive version *P′*, technique *M*, and test suite *T*, the following were measured:

1. The proportion of test cases in the selected test suite *T′* to test cases in the original suite *T*.
2. The number of MPs revealed by *T* and *T′*.

From these data points two dependent variables were computed:

1. Average selected test suite size.
2. Average MP detection effectiveness.

Threats to Validity. In this study, the main concern for threats to internal validity is that instrumentation effects can bias the results. Instrumentation effects may be caused by differences in the experimental instruments (in this case the test process inputs: the code to be tested, the locality of the program changes, the composition of the test suite, or the composition of the series of versions). In contrast to Experiment 1, all modifications to DNAM are naturally occurring, not seeded. On the other hand, the test suites do not exercise large parts of the program. Since it was not possible to create new test cases, it was not possible to control for this situation. It was also not possible to control the number of changes made between two versions, or the number of versions with a specific number changes made to them.

For threats to external validity, the greatest concern involves the representativeness of the program. This threat is partially overcome in this study since DNAM is a real project that is used in the field, with real changes incorporated over time by user demand. In addition, the program is

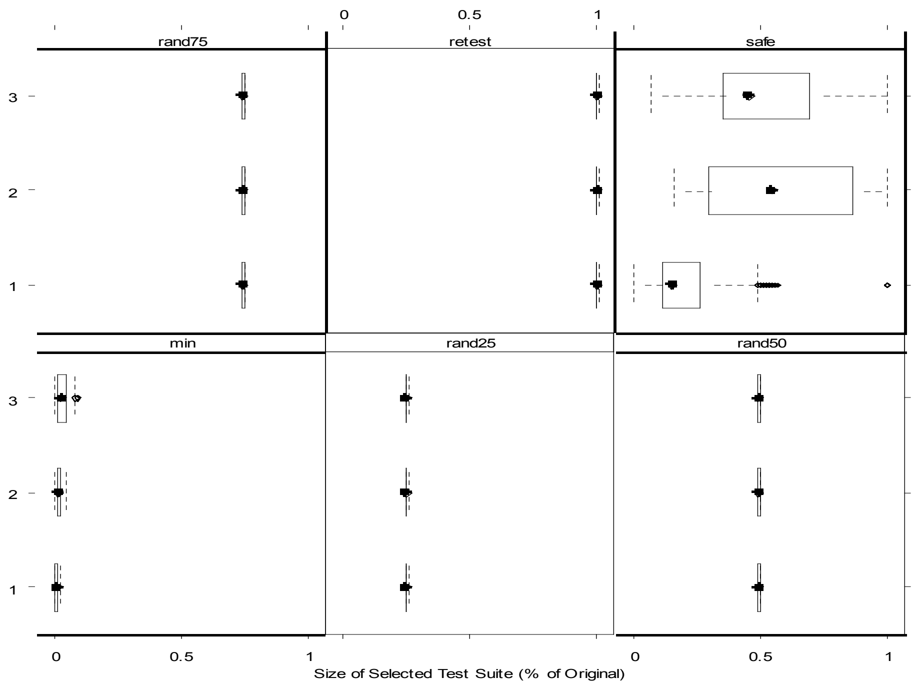

Figure 6: Test Suite Size by Change Interval

quite large. Nevertheless, DNAM is just one program and does not represent all kinds of programs so further studies will be needed. Another threat to external validity for this study is process representativeness. This arises when the testing process used is not representative of industrial ones. This may endanger results since the test suites used may be more or less comprehensive than those that could appear in practice. Also, this experiment mimics a corrective maintenance process where the specification is not changed, but there could be many other types of maintenance in which regression testing might be used. These threats can be addressed only through additional studies using a greater range of software artifacts.

4.5 Data and Analysis

Size Reduction. Figure 6 shows the ability of each RTS technique to reduce test suite size, by interval, conditioned on the technique itself. Because the data at various settings of the interval is sparse, the program changes are grouped into three categories: 1-3 MPs, 4-7 MPs, and 8 or more

MPs. Each of these categories contains roughly one third of the data points. The random(*n*) methods selected *n*% of the test cases by construction and the retest-all method always selected all test cases.

For the safe technique, the selection ratio appears to grow as the interval grows. The median selection ratio for interval group-1 was 15.58%, while it increased to 54.55% for interval group-2 and 45.64% for interval group-3. This is consistent with the hypothesis that the selection ratio is affected by the number of modifications made between the testing sessions.

For minimization, the selection ratio for DNAM increases as the interval grows. The median overall selection ratio for DNAM is 1.96%. Again, this pattern is consistent with that seen in Experiment 1. There, the median selection ratio went from 0.4% for interval 1 to 2.0% for interval 10. In Experiment 2, the median selection ratio was 1.31% for interval Group-1, increasing to 3.27% for interval Group-3.

Relative Effectiveness. Figure 7 shows the relative effectiveness of each interval setting conditioned on technique. As the safe technique guarantees that all detectable MPs will be detected, the analysis concentrates on the random and minimization methods.

Overall, the trends for each technique coincide with those of Experiment 1. First, for random methods, the random(*n*) technique became more effective on the average as *n* increased. Moreover, their performance was not greatly affected by interval.

The performance of minimization also displayed trends similar to those found in Experiment 1. Specifically, effectiveness increased substantially as the interval grew (median relative effectiveness of 42.77% for group-1, 47.66% for group-2, and 84.15% for group-3).

5 COMPARING THE TWO STUDIES

The results of two empirical studies of regression test application frequency have been presented.

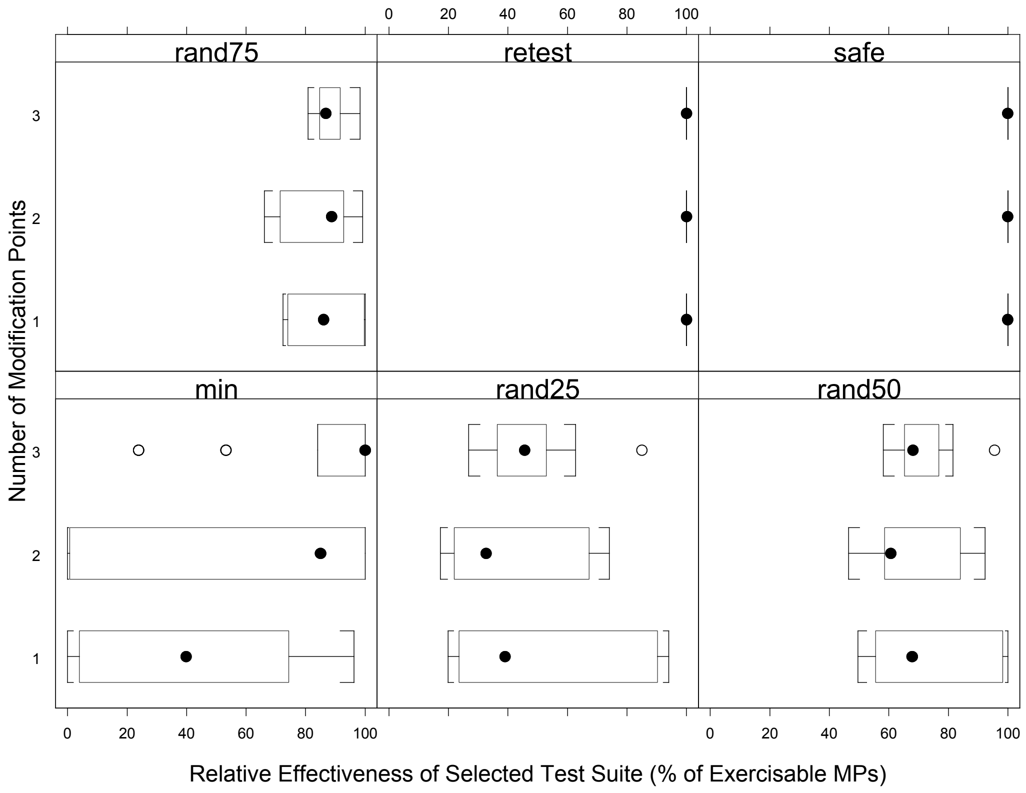

Figure 7: Test Suite Effectiveness by Change Interval

Some of the costs and benefits of several RTS techniques when the number of changes between the base and subsequent versions of a program increases have been investigated. The results of this investigation highlight several differences among RTS techniques with respect to test application frequency. They also illustrate some tradeoffs and provide an infrastructure for further research.

As discussed earlier, these studies, like any controlled experiments, have several limits to validity. Particularly, several threats to external validity limit the ability to generalize results. These threats can be addressed only by extensive experiments with a wider variety of programs, test suites, series of versions, type of faults, and so forth. Along these lines, however, it is revealing that the second study, conducted on a large-scale program, appears to support the results of the first study, which was performed on much smaller programs. Keeping all this in mind, the following conclusions can be drawn.

The experimental results strongly support hypotheses H1 and H2: the size and fault detection effectiveness of test suites chosen by RTS techniques change as the frequency of regression testing changes.

Safe techniques selected the same as or fewer test cases than retest-all while having the same effectiveness. Therefore, they are preferable to retest-all as long as their analysis costs are less than the cost of running the unselected test cases. However, as the testing interval grew, almost all test cases were selected for the Siemens/Space programs. In this case, safe methods may not be preferable to retest-all. This effect was less dramatic in the case of DNAM, but it could be seen when comparing smaller changes (1-3 MP's) to larger changes (4 or more MP's).

Nevertheless, the analysis also showed, reflecting results of other studies [16], that the performance of safe techniques depended heavily on the structure of the program, the location of modifications, and the composition of the test suites. For instance, test cases for the program `schedule2` are constructed in such a way that each test case exercises large portions of the code. Consequently, changes, no matter how small, tend to involve all test cases in the test suite. Test cases that exercise independent portions of the system might not exhibit such behavior and thus might be more amenable to safe test selection. For example, in DNAM, both the test cases and changes were concentrated in a number of libraries. This appears to have limited test suite size. Research that successfully merges RTS techniques with test suite construction is likely to have a large effect on the use of safe techniques in practice.

Random techniques are surprisingly cheap and effective. Interestingly, as the testing interval increases their median effectiveness approaches that of retest-all with less variation between runs. That is: at small testing intervals effectiveness ranges from very high to very low. However, as the testing interval increases, this range gets much smaller. Thus, a user of random techniques might be more confident of their effectiveness in the latter situation.

The difference in performance between minimization at low testing intervals and minimization at high testing intervals was remarkable for all programs. For the Siemens/Space programs, at low testing intervals, minimization selected one or two test cases and was only 16% as (relative) effective as retest-all. However, at a testing interval of 10 it selected only four or five test cases, while having about 60% of the (relative) effectiveness of retest-all. For DNAM, at low intervals, minimization selected 0.4% of the test cases and was 44.8% as effective as at high intervals. Yet at the largest testing interval it selected only 3.27% of the test cases, but had an 84.15% relative effectiveness. Thus, although the minimization approach will miss some faults, from a cost-benefit perspective it presents an interesting option. It is not clear why this effect is so pronounced, and continued investigation is needed. Also, in this study minimization picks exactly one test case through a change. It would be interesting to investigate what would happen if it instead picked two, three or more.

As testing intervals increase, complex fault interactions can make it harder to detect some faults if the code is monolithic. This is somewhat obvious. More to the point, however, is that as the testing interval increases, information about a base program P and test suite T will become less predictive of the state of P'. For example, a test suite that was edge-coverage-adequate on program P might have very different coverage of program P'.

6 CONCLUSIONS AND FUTURE WORK

Making progress in understanding regression test selection techniques depends, not on a single empirical study, but on entire families of studies, including both controlled experiments and case studies. By replicating the earlier study of application frequency on a larger, more complex program, this work has begun to construct such a family. The experimental designs utilized can also be followed by other researchers seeking to address the open questions in this area.

Work is ongoing to continue this family of experiments. Goals include (1) improving cost models to include factors such as testing overhead and to better handle analysis cost, (2) extending the experiment to larger programs with a wider variety of naturally-occurring faults, and (3) exploring techniques that incorporate previous testing history.

ACKNOWLEDGEMENTS

This work was supported in part by the following grants from the National Science Foundation CCR-9501354, CCR-9707792, CCR-0205265 and CCR-0098158. Siemens Laboratories supplied several subject programs. Alberto Pasquini, Phyllis Frankl, and Filip Vokolos provided the Space program and many of its test cases. Chengyun Chu assisted with further preparation of Space program and its test cases. Will Gillett at Washington University, Saint Louis provided the DNAM program and its test cases.

REFERENCES

1. H. Agrawal, J. Horgan, E. Krauser, and S. London. Incremental regression testing. In Proceedings of the Conference on Software Maintenance, pages 348-357, Sept. 1993.
2. 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 211-222, May 1994.
3. 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.

4. T.L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel, An empirical study of regression test selection techniques. ACM Transactions on Software Engineering and Methodology, 10(2): pp. 184-208 (2001).

5. M.J. Harrold and M.L. Soffa. An incremental approach to unit testing during maintenance. In Proceedings of the Conference on Software Maintenance, pages 362-367, Oct. 1988.

6. J. Hartmann and D. Robson. Techniques for selective revalidation. IEEE Software, 16(1):31-38, Jan. 1990.

7. 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 16th International Conference on Software Engineering, pages191-200, May 1994.

8. H.K.N. Leung and L.J. White. A cost model to compare regression test strategies. In Proceedings of International Conference on Software Maintenance, pages 201-208. Oct. 1991.

9. T. Ostrand and M. Balcer. The category-partition method for specifying and generating functional tests. Communications of the ACM, 31(6), June 1988.

10. T. Ostrand and E. Weyuker. Using dataflow analysis for regression testing. In Sixth Annual Pacific Northwest Software Conference, pages 233-247, Sept. 1988.

11. J. Bible, G. Rothermel, and D. Rosenblum, A comparative study of coarse- and fine-grained safe regression test selection. ACM Transactions on Software Engineering and Methodology, V. 10, no. 2, April, 2001, pages 149-183.

12. D. Rosenblum and E. J. Weyuker. Lessons learned from a regression testing case study. Empirical Software Engineering Journal, 2(2), 1997.

13. G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. IEEE Transactions on Software Engineering, 22(8):529-551, Aug. 1996.

14. G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. ACM Transactions on Software Engineering and Methodology, 6(2):173-210, Apr. 1997.

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

16. G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. IEEE Transactions on Software Engineering, 25(6), pages 401-419, June 1998.

17. F. I. Vokolos and P. G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In Proceedings of the International Conference on Software Maintenance, pages 44-53, Nov. 1998.

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

19. J. M. Voas, PIE: A dynamic failure-based technique, IEEE Transactions on Software Engineering, August 1992, 18(8), pp. 717-727.

20. J.-M. Kim, A. Porter, G. Rothermel, An empirical study of regression test application frequency. In Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland, June 2000, pages 126-135.

21. A. Malishevsky, S. Elbaum, G. Rothermel, Modeling the cost-benefits tradeoffs for regression testing techniques. In Proceedings of the International Conference on Software Maintenance, Oct. 2002, pages 204-213.

22. S. Elbaum, P. Kallakuri, A. G. Malishevsky, G. Rothermel, and S. Kanduri, Understanding the effects of changes on the cost-effectiveness of regression testing techniques. Journal of

Software Testing, Verification, and Reliability., V. 13, no. 2, June, 2003, pages 65-83.