# A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments

Jung-Min Kim Adam Porter
Department of Computer Science
University of Maryland, College Park
College Park, MD 20742, USA
+1-301-405-2702
{jmkim,aporter}@cs.umd.edu

## ABSTRACT

Regression testing is an expensive and frequently executed maintenance process used to revalidate modified software. To improve it, regression test selection (RTS) techniques strive to lower costs without overly reducing effectiveness by carefully selecting a subset of the test suite. Under certain conditions, some can even guarantee that the selected test cases perform no worse than the original test suite.

But this ignores certain software development realities such as resource and time constraints that may prevent using RTS techniques as intended (e.g., regression testing must be done overnight, but RTS selection returns two days worth of tests). In practice, testers work around this by prioritizing the test cases and running only those that fit within existing constraints. Unfortunately this generally violates key RTS assumptions, voiding RTS technique guarantees and making regression testing performance unpredictable.

Despite this, existing prioritization techniques are memoryless, implicitly assuming that local choices can ensure adequate long run performance. Instead, we proposed a new technique that bases prioritization on historical execution data. We conducted an experiment to assess its effects on the long run performance of resource constrained regression testing. Our results expose essential tradeoffs that should be considered when using these techniques over a series of software releases.

## Keywords

Regression testing, test history, prioritization, empirical study

## 1. INTRODUCTION

After modifying software, developers typically want to know that unmodified code has not been adversely affected. When such unmodified code is adversely affected, we say that a *regression error* has occurred. Developers often do *regression testing* to search for such regression errors. The simplest regression testing strategy is to rerun all existing test cases. This method is simple 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) technique, has been proposed (e.g., [1],[3],[6],[7],[13],[17]). With this approach only a subset of test cases are selected and rerun. Since, in general, optimal test selection (i.e., selecting exactly the fault-revealing test cases) is impossible, the cost-benefit tradeoffs of RTS techniques are a central concern of regression testing research and practice.

Our understanding of these techniques, however, is limited. One reason is that researchers commonly study this problem by finding or creating base and modified versions of a system and accompanying test suites. Next, they run test selection algorithms and compare the size and effectiveness of the selected test suite to the size and effectiveness of the original test suite (e.g., [5],[14],[19],[21]). There are two critical limitations to this approach: (1) it models regression testing as a one-time activity rather than as the continuous process it is, and (2) it does not take real world time and resource constraints into consideration. The fact is that regression testing is far better modeled as an ordered sequence of testing sessions, each of whose performance may depend upon prior testing sessions, and each of which is subject to time and resource constraints.

From these observations two conjectures flow. One is that there can be a big difference between what an ideal regression tester should do and what the actual one can afford to do. For example, a recent study of ours [9] suggests that the amount of changes made between testing sessions strongly affects the performance of different RTS techniques. In fact, one approach, called a safe technique [17], routinely selected almost all test cases when more than a few (2 or 3) changes were made to the subject programs. If this situation arises in practice (like when a system is undergoing heavy modifications in a time-constrained development environment), then RTS techniques can't be used as intended. Instead the RTS-selected test cases must be reduced even further so they can be executed under given constraints. This is called test case prioritization (e.g., [20],[22]).

A second conjecture is that historical test case performance data, which current RTS and test case prioritization techniques ignore entirely, might be used to improve long run regression testing performance. Current RTS and prioritization techniques are memoryless. They are based only on the analysis of source code and test case profiling information taken from the current and immediately preceding software versions. This implicitly assumes that local information can ensure adequate long run performance. But prioritization voids RTS technique guarantees, making regression testing performance unpredictable. Consequently, in situations where we must resort to test case prioritization, we must admit the possibility that new techniques not based solely on local information might have merit.

In this article, we begin exploring these conjectures. In particular, we evaluate how several RTS techniques perform under severe time and resource constraints. We also present and evaluate one simple heuristic that uses historical information to do test case prioritization. We hypothesize that such heuristics can, over the long run, reduce the cost and increase the effectiveness of regression testing in constrained development environments. If this hypothesis is true, testing practitioners may be able to better manage and coordinate their integration and regression testing processes, thereby saving time and money. Therefore we have designed and implemented an experiment to examine this hypothesis.

In the remainder of this paper we review the relevant literature, describe our research hypotheses, present the design and analysis of our experiment and discuss our conclusions and future research directions.

## 2. BACKGROUND

### 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 $T'$. Establish $P'$'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 is important. However, we restrict our attention to step 1 - the *regression test selection problem*.

### 2.2 RTS Techniques

Several regression test selection techniques have been investigated in the literature (see [16]). Here we briefly describe several techniques and give a representative example of each.

#### *2.2.1 Retest-All Technique*

This method reruns all test cases in $T$. It may be used when test effectiveness is the utmost priority with little regard for cost.

#### *2.2.2 Random/Ad-Hoc Technique*

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 $T$.

#### *2.2.3 Minimization Technique*

This approach (e.g., [4], [7]) aims to select a minimal set of test cases from $T$ that covers all modified elements of $P'$. One such technique randomly selects test cases from $T$ until every program statement added or modified to create $P'$ is exercised by at least one test case.

#### *2.2.4 Safe Technique*

These techniques (e.g., [3],[17]) select, under certain conditions, every test case in $T$ that covers changed program entities in $P'$. One such technique selects every test case in $T$ that exercises at least one statement that was added or modified to create $P'$, or that has been deleted from $P$.

### 2.3 Leung and White's Cost Model

Leung and White [10] present a simple 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 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 regression test selection strategy $M$ 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 running the unselected tests, $T - T'$.

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.

### 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 [15] applied their technique to 31 versions of the KornShell and its test suites. Rothermel and Harrold [17] conducted a similar study with their technique, using several 100- to 500-line programs and a larger (50 KLOC) program. These two studies seem to indicate that in some cases, regression test selection can be cost-effective. Later studies, therefore, begin to compare different methods.

Rosenblum and Rothermel [14] compared the performance of two safe techniques in terms of test selection. The study, however, did not compare other techniques nor consider fault detection.

Graves et al. [5] examined the costs and benefits of several regression test selection 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 researchers drew the following overall conclusions:

Some program analysis based (PAB) techniques (e.g., safe and dataflow) were effective in detecting faults, but the variance in the number of test cases selected was quite large.

Equally sized, randomly selected test suites were nearly as effective as PAB techniques.

Minimization yielded the smallest and the least effective test suites.

Kim et al. [9] investigated how the number of changes made between base and subsequent versions affected the performance of several RTS techniques. They drew the following conclusions:

The percentage of test cases selected by safe RTS techniques grew to almost 100% when as few as 3 changes were made.

Random selection was surprisingly cheap and effective and its effectiveness was not greatly affected by change activity.

Minimization selected very few tests and became much more effective as the number of changes increased.

Profile data on *P* using *T* became much less predictive of execution behavior as the number of changes grew.

Some work has also been done to study test case prioritization. Wong et al. [22] proposed several techniques: (1) modification-based test selection then block-coverage-preserving minimization and (2) modification-based test selection then prioritization based on the increasing order of additional cost per coverage. They conducted a case study in which their techniques were applied to a 5000 line program with ten faulty versions. They concluded that both techniques could be cost-effective alternatives in constrained environments.

Rothermel et al. [20] also proposed and evaluated a family of prioritization techniques. Based on several different programs and test suites, their study suggested that their techniques could improve fault detection rate (faults/number of test cases run). It also suggested that more expensive techniques might not be as cost-effective as other less expensive techniques.

Both of these efforts involved memoryless prioritization techniques and modeled regression testing as a one-time activity ignoring possible effects across multiple software releases. Finally, neither took time or resource constraints into consideration. This leads us to consider several open questions.

## 2.5 Open Questions

In this research, we consider three facets of RTS:

The test selection technique,

The application policy - the conditions that trigger regression testing: periodic execution (daily, weekly, or monthly), or rule-based execution (after all changes, after changing critical components, or at final release), and

Process factors such as resource constraints and deadline - when regression testing is done in constrained environments developers may have to limit their testing efforts.

Most previous studies have focused on the first facet while ignoring the second and third. Yet, these latter facets are important because they may greatly affect the practical costs and benefits of regression test selection.

We recently studied the second facet [9], showing that it strongly affects RTS costs and benefits. This paper continues that line of research, investigating how process factors such as time and resource constraints affect the regression testing process. In particular, we focus on test case prioritization techniques. Our goal is to see whether basing test case prioritization on historical data affects the long-term performance of regression testing done in constrained environments.

## 3. EVOLUTION MODELS

Previous studies model regression testing as a set of unordered, independent testing sessions. Regression testing is far better modeled as an ordered sequence of testing sessions, each of which may be dependent on the previous testing sessions. Ignoring this distinctions risks:

Losing important information. For example, minimization techniques focus testing on parts of the program that have changed since the last testing session. So it is possible that a change, once tested, is never re-tested. If that change contains a fault, we have only one chance to find it. Unless we consider interactions between testing sessions, we won't uncover those kinds of situations.

Misinterpreting results. Our previous research shows that regression testing frequency affects performance. Existing studies haven't considered this issue, which severely limits the applicability of their results.

Missing improvement opportunities. Regression testing generates huge amounts of data that are currently ignored. Analysis of this data might reveal dependencies that can be exploited. For example, such analysis might uncover groups of test cases that perform similarly (have correlated pass/fail behavior), allowing test suites to be pruned.

Thus we believe that there is an overwhelming need for more rigorous and more realistic regression testing models. In this paper, we use two different models of regression testing.

### 3.1 Model 1

Here we model an evolving software system: $P_0, P_1, \ldots P_{n-1}, P_n$, where $P_0$ is the base version, and $P_{i+1}$ is $P_i$ with a single change applied. With our available subject programs, each single change is faulty. Our regression test process starts with versions $P_0$ and $P_1$, then $P_1$ and $P_2$. The process continues until all the faults inserted are detected, (but no new changes are added after $P_n$.)

### 3.2 Model 2

Here we model a fault removal process: $P_0, P_1, \ldots P_{n-1}, P_n$, where $P_0$ contains all existing faults, $P_{i+1}$ is the subsequent version of $P_i$ after applying some RTS technique and then removing any identified fault(s). This process continues until all known faults are detected.

## 4. TEST CASE PRIORITIZATION

Until now researchers have assumed that developers could, if necessary, rerun all test cases in a single testing sessions. As we have said, this is not always possible. For example, if we want to regression test every night, then compilation and testing time must take less than, say 8-10 hours. Also, regression testing of embedded systems is often done using sophisticated simulation environments. Such environments are expensive and, as they are usually shared by multiple projects, access to them is very limited. Current RTS techniques are oblivious to these constraints and, thus, may select more test cases than can be run in a given testing session. In such cases we must find a way to further reduce the selected test suite.

At a high level, test case prioritization works as follows: (1) apply an RTS technique to test suite *T*, yielding $T'$, (2) assign a selection probability to each test case in $T'$, (3) draw a test case from $T'$ using the probabilities assigned in step 2, and run it, and (4) repeat step 3 until testing time is exhausted.

The key question is how to set/assign the selection probabilities. Our idea is to use information about each test case's prior

**Table 1: Experimental Subjects**

| Program Name | Functions | LOC | # of Versions | Test Pool Size | Avg. Test Suite Size |
|---|---|---|---|---|---|
| replace | 21 | 516 | 15 | 5542 | 226 |
| printtokens | 18 | 402 | 5 | 4130 | 128 |
| printtokens2 | 19 | 483 | 8 | 4113 | 65 |
| schedule | 18 | 299 | 8 | 2650 | 107 |
| schedule2 | 16 | 297 | 8 | 2710 | 113 |
| tcas | 9 | 138 | 18 | 1600 | 84 |
| totinfo | 7 | 346 | 18 | 1052 | 92 |
| space | 136 | 6218 | 30 | 5179 | 80 |

performance to increase or decrease the likelihood that it will be used in the current testing session. Our approach is based on ideas taken from statistical quality control (exponential weighted moving average) and statistical forecasting (exponential smoothing)[2].

We define the selection probabilities of each test case *tc* at time *t* to be $P_{tc,t}(H_{tc}, \alpha)$, where $H_{tc}$ is a set of *t*, time-ordered observations $\{h_1, h_2, \ldots, h_t\}$ drawn from previous runs of *tc*, and $\alpha$ is a smoothing constant used to weight individual history observations (higher values emphasize recent observations, while lower values emphasize older ones). These values are then normalized and used as probabilities. The general form of P is:

$$P_0 = h_1$$

$$P_k = \alpha h_k + (1 - \alpha)P_{k-1},\ 0 \leq \alpha \leq 1,\ k \geq 1$$

Different test histories (definitions of $H_{tc}$) will yield different prioritizations. For example, we are investigating test histories based upon each test case's execution history, its fault detection, and/or the program entities it covers.

Based on test execution history. For every testing session *i,* in which test case *tc* is executed, $h_i$ takes the value 0. Otherwise it takes the value 1. If we use a low value of $\alpha$ this will assign higher selection probabilities to test cases that have not been run recently and lower ones to those that have. The net effect is to cycle through all test cases over multiple testing sessions.

Based on demonstrated fault detection effectiveness. For every testing session *i,* in which test case *tc* passed, $h_i$ takes the value 0. Otherwise it takes the value 1. If we use a high $\alpha$-value, we will assign high selection probabilities to test cases that have revealed faults recently and low weights to those that have not. One effect should be to limit the running of test cases that rarely, if ever, reveal faults. Another should be that test cases whose failures are related to unstable sections of code would continue to be selected until that code stabilizes.

The third approach is based on the coverage of program entities. Program entities might include statements, paths, functions, def-use pairs, etc. Without loss of generality, let's use functions as the program entity of interest. Our goal is to give higher priority to test cases that cover functions infrequently covered in past testing sessions. Thus, for every testing session, *i*, we execute the following two steps. First, we assign a weight to each function such that infrequently covered functions have much higher weights than frequently covered ones. Specifically, we calculate the number of test cases that covered each function. Then we weight each function such that its weight has an inverse exponential relationship to the number of test cases that covered it and such that the total weights over all functions sums to 1. Second, we define the test history, $H_{tc}$. Specifically, $h_i$ is the sum of the weights for all functions that *tc* covered. If we use high $\alpha$-value, we will assign high selection probabilities to test cases that cover functions not recently exercised. The net effect would be to limit the possibility that any particular function goes unexercised for ling periods of time.

## 4.1 Constraint-aware Prioritization Methods

For this article we have implemented a prioritization method based on test execution history and we compare it against two controls:

1. Lru(n): This approach uses the test execution history described in the previous section, with $\alpha$ set as close to 0 as possible (actual value is machine dependent). Additionally we assume that time constraints allow us to execute only n% of the original test suite. Therefore, *lru(n)* chooses the n% of the test suite with the highest selection probabilities. One of the virtues of this method is that it cycles through all test cases over multiple testing sessions.

2. Safe-random(n): As a control we examined another approach called *safe-random(n).* This approach starts by using a safe RTS technique. If the number of test cases selected by the safe technique is greater than the limit (n% of original test suite), then, from this set, we select the appropriate number of test cases on a random basis. The rest are timestamped and saved in a repository. If, instead, the number of test cases selected by the RTS technique is fewer than the limit, then we use the entire selection, and add test cases randomly selected from the repository.

3. Random(n): Randomly select n% of the original test suite.

## 5. THE EXPERIMENT

## 5.1 Hypotheses

We hypothesize that history-based test prioritization methods help to reduce the cost and to increase the effectiveness of regression testing process in the long run.

## 5.2 Measures

To investigate our hypothesis we need to measure the costs and benefits of each test selection and prioritization technique. To do this we constructed two models: one for calculating savings in terms of total efforts, and another for calculating costs in terms of age of fault. We restrict our attention to these costs and benefits, but there are many other costs and benefits these models do not capture. Some other costs and benefits are mentioned in Section 0.

### *5.2.1 Measuring Total Effort*

Reducing test suite size saves time because we run fewer test cases, examine fewer test results, and manage less test data. In our experiment we used each RTS technique until all faults were detected or we reached 50 testing sessions. We then summed the number of test cases run across all testing sessions. This allowed us to measure the *total effort* expended across all sessions.

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., we don't differentiate between CPU time and human effort). It also does not measure the savings that may result from reusing analyses done during early testing sessions during later testing sessions.

#### *5.2.2 Measuring Fault Age*

We considered two types of costs. 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. Our cost model focuses on the latter cost, assuming that regression test selection is cost-effective under the definition given by Leung and White (see Section 2.3).

To determine whether a given test selection approach reduces fault detection effectiveness we need to know which test cases reveal which faults in $P'$. Because this information is difficult to obtain, we estimate it in the following way [9]. At the end of each testing session we determine which faults were identified.

> In order for a test case $t$ to detect a fault $f$, three conditions must be satisfied: (1) $t$ must traverse the program point containing $f$ in $P'$, (2) immediately after $t$ traverses the program point containing $f$ in $P'$, key 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 $t$.

If a given fault was not identified we increment a counter associated with that fault. Testing continues until either all the faults have been detected and removed or until 50 testing sessions have been conducted. The value of these counters at the end of the testing process is called the *fault age* of that fault.

## 5.3 Experimental Instrumentation

#### *5.3.1 Programs*

For our study, we obtained eight C programs with a number of modified versions and test suites for each program. The subjects come from two sources. One is a group of seven C programs collected and constructed initially by Hutchins et al. [8] 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. We slightly modified some of the programs and versions in order to use them with our tools. 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. We describe these and other data in the following paragraphs.

**Siemens Programs:** Seven of our subject programs come from a previous experiment done by Hutchins et al. [8]. These programs are written in C, and range in size from 7 to 21 functions and from 138 to 516 lines of code.

For each of these programs Hutchins et al. created a pool of black-box test cases [8] using the *category partition method* and Siemens Test Specification Language tool [12]. 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" ([8], 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 30 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 we obtained a pool of randomly generated test cases created by Vokolos and Frankl [21]. Then we added new test cases until every dynamically executable edge[1] in the program's control flow graph was exercised by at least 30 test cases.

#### *5.3.2 Versions*

In this experiment program versions needed to contain several faults at the same time. To do this, we identified "*mutually independent*" faults. That is faults that could be automatically merged into the base program without interfering with each other. 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 mutually independent versions for each subject program, ranging from 5 to 30.

#### *5.3.3 Test Suites*

We used test pools to obtain augmented edge-coverage-adequate test suites for each program. To do this, we took the test pool for the base program and its associated test coverage information and used it to generate 1000 edge-coverage-adequate test suites for each base program. Then we augmented each test suite, making sure that the test suite contained at least one fault revealing test case for each fault. This augmentation prevents us from confusing an inadequate test quite with an ineffective test selection and prioritization approach.

#### *5.3.4 Test Selection Techniques and Tools*

To perform the experiments, we needed implementations or simulations of regression test selection tools. For the *safe*

[1] Excluding those edges that can be exercised only by the occurrence of malloc faults.

technique we used an implementation of Rothermel and Harrold's **DejaVu** tool [18]. For *minimization*, we created a tool that selects a minimal test suite *T′* such that *T′* has at least one test case that covers every node in the control flow graph for *P* that was changed between *P* and *P′*. For the *random(n)* technique we created a tool that randomly selects *n%* of the test cases from the suite. We implemented our own *lru(n)* technique. To do this we needed to save the test history from each regression testing session. We implemented the *safe-random(n)* technique by first calling the safe technique. This returns a set of test cases that we call the selected test cases. Our goal is to run *x* test cases, where *x* is equal to *n*% of the original test suite. If there are more than *x* test cases in the selected test suite, then we randomly select *x* of them and place the rest into a repository. Otherwise, we use all selected test cases and then add some more from the repository until we have selected a total of *x* test cases. *Retest-all* does not require any tools.

## 5.4 Experimental Design

### 5.4.1 Variables

The experiment manipulated three independent variables:

1. The subject program (there are 8 programs, each with a variety of modified versions).
2. The test selection technique (one of safe, minimization, retest-all, random(5), random(10), random(20), lru(5), lru(10), lru(20), safe-random(5), safe-random(10), and safe-random(20)).
3. Two different evolution models (see Section 3).

For each combination of program and technique we applied 100 augmented edge-coverage-adequate test suites. On each test run, with base program *P*, modified version *P′*, technique *M*, and test suite *T*, we measured:

1. The number of test cases in the selected test suite *T′*.
2. The number and identity of faults revealed by *T* and *T′*.

From these data points we computed two dependent variables:

1. Total testing effort.
2. Average fault age.

The experiment used a full-factorial design with 100 repeated measures. That is, for each subject program we selected 100 test suites from the test suite universe. For each test suite, we then applied each test selection technique and measured the size and fault detection effectiveness of the selected test suites. In total, the experiment required us to run nearly 4,000,000 test cases.

### 5.4.2 Threats to Validity

In this section we consider some of the potential threats to the validity of our study.

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 our study, our greatest concern is that instrumentation effects can bias our 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 our subject programs are considered faults. In reality, some modifications will not result in faults. In this study we used augmented edge-coverage-adequate test suites. However, at this time we do not control for the structure of the subject programs, or for the locality of program changes. To limit problems related to this, we run our test selection algorithm on each suite and each subject program.

Threats to external validity are conditions that limit our ability to generalize the results of our 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 every effort to make them as realistic as possible was taken. 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. 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 we used is not representative of industrial ones. This may endanger our results since the test suites we utilized may be more or less comprehensive than those that could appear in practice. Also, the modifications we make do not change the program specification. In practice, this does happen. We have tried to allow for different kinds of evolution by using two different software evolution models. These threats can only be addressed through additional studies using a greater range of software artifacts.

## 6. DATA AND ANALYSIS

In this paper, we use box plots (e.g., Figure-1) 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 "outliers".

## 6.1 Model 1

Model-1 involves a correct base version with testing after each new change.

### 6.1.1 Fault Age

Figure-1 is a box plot showing the distribution of fault age for each RTS technique under Model-1. Table-2 shows the median, average and standard deviation of fault age for each RTS technique under Model-1.

As each test suite is augmented edge-coverage-adequate, the *retest-all* and *safe* techniques immediately detect each fault in each version. The other techniques sometimes missed faults, which passed into subsequent versions, raising average fault age.

*Minimization* had the highest median and average fault age and the largest standard deviation.

One interesting observations is that the standard deviation of fault age for the *lru(n)* techniques is less that that for the other techniques. This is because the *lru(n)* techniques found all faults well before the 50 testing session cutoff. In contrast, the *minimization*, *safe-random(n)* and *random(n)* techniques allowed

**Table-2: Median, Average and Standard Deviation of Fault Age by RTS Technique (Model-1).**

| | median | average | std. dev. |
|---|---|---|---|
| safe | 1 | 1 | 0 |
| retest-all | 1 | 1 | 0 |
| min | 7 | 11.0 | 13.0 |
| lru(5) | 6 | 7.4 | 6.3 |
| lru(10) | 4 | 5.2 | 4.9 |
| lru(20) | 2 | 3.5 | 3.8 |
| rand(5) | 5 | 8.7 | 10.0 |
| rand(10) | 3 | 5.8 | 6.8 |
| rand(20) | 2 | 3.7 | 4.5 |
| safe-rand(5) | 3 | 6.5 | 9.1 |
| safe-rand(10) | 2 | 3.9 | 5.4 |
| safe-rand(20) | 1 | 2.5 | 3.8 |

faults to go undetected over 50 testing sessions. Moreover, since the cutoff is simply an artifact of the experiment, it is likely that the faults would persist even longer in practice.

We now compare the prioritization methods in more detail.

**LRU vs. Random.** For equal values of *n*, the median fault age of *random(n)* is slightly lower[2] than that of *lru(n)*. The mean and standard deviation, however, are lower for *lru(n)*. Note that the "outliers" for *random(5)* and *random(10)* reach the cutoff value of 50, implying that these statistics might be higher in practice.

**Random vs. Safe-Random.** For equal values of *n,* the fault age for *safe-random(n)* had a lower median, average, and standard deviation *random(n)* did. Both techniques, however, have outliers at 50 testing sessions, indicating that in some cases faults went completely undiscovered.

**LRU vs. Safe-Random.** For equal values of *n*, *safe-random(n)* performed better than *lru(n)* in terms of median and average fault age. However, the standard deviation is lower for the *lru(n)* technique. Also, the difference in standard deviation increases as n decreases.

### *6.1.2 Total Effort*

Figure-2 contains a box plot showing the distribution of total effort across different RTS techniques.

Total effort is defined as 100 times the proportion of the total number of test cases executed by a given technique to the total number of test cases executed by retest all. For example, the Space program has 30 versions and its average test suite has 80 test cases. Retest-all detects all faults after 30 regression test sessions, using 2400 test cases in total. Let's assume that a given

[2] We say that one method performed better (worse) than another only if such a statement is supported by a t or wilcoxon test with p < 0.5. Qualifiers like *slightly* or *substantially* are subjective.

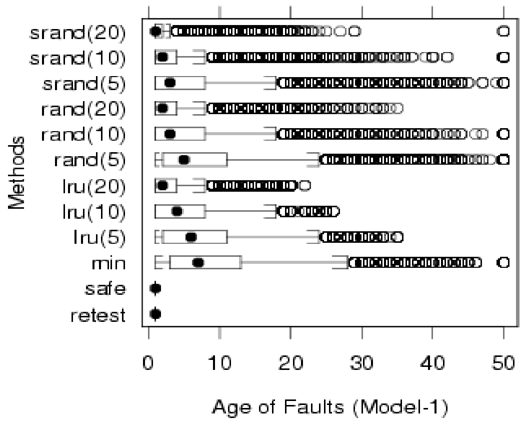

**Figure-1: Fault Age by RTS Technique (Model-1).**

experimental run applied *lru(5)* to the Space program, involved 4 test cases per testing session, and required 40 sessions to identify all faults. Here the total effort would be 5 — 100 times 120 (the total number of test cases executed across all runs of *lru(5)*) divided by 240 (the total executed by retest-all). Table-3 shows the median and average total efforts for each different RTS techniques for Model-1.

We draw several observations from this data. The total effort for the *safe* technique is less than that of *retest-all*, although the variance (not shown) is quite large. At least for these programs this is consistent with our earlier observation that safe techniques may be difficult to use in constrained environments.

We also see that the total effort of *minimization* is very low (median 5.1). The tradeoff, as we saw in the previous section, is that it takes longer to find faults this way. (median fault age 7).

Finally, all methods besides *retest-all* required less total effort than the *safe* technique.

We now compare the prioritization methods in more detail.

**LRU vs. Random.** For equal values of *n*, *lru(n)* required less total effort than *random(n)* both in terms of the median and the average.

**Random vs. Safe-Random.** For equal values of *n, safe-random(n)* required less total effort than *random(n)* both in terms of the median and the average.

**LRU vs. Safe-Random.** For equal values of *n, safe-random(n)* performed slightly better than *lru(n)* in terms of median and average total efforts.

### *6.1.3 Cost-Benefit Tradeoffs*

**Safe.** The operating assumption in this paper is that we are in a constrained environment that prevents us from using unmodified safe techniques (or *retest-all*). In this study, we saw that the average safe test suite was roughly 60% as large as the original test suite (although it varied considerably). These data are

**Table-3: Median and Average Total Effort by RTS Technique under Model-1.**

| | median | average |
|---|---|---|
| safe | 61.3 | 60.1 |
| retest-all | 100 | 100 |
| min | 5.1 | 5.7 |
| lru(5) | 12.1 | 12.7 |
| lru(10) | 14.3 | 16.2 |
| lru(20) | 25.0 | 26.5 |
| rand(5) | 15.6 | 17.7 |
| rand(10) | 20.4 | 23.2 |
| rand(20) | 28.3 | 31.4 |
| safe-rand(5) | 9.9 | 11.4 |
| safe-rand(10) | 13.6 | 15.8 |
| safe-rand(20) | 22.2 | 25.0 |

consistent with our conjecture that in some cases unmodified *safe* techniques may be inappropriate (although this is obviously situation dependent). It is also interesting to note that all other techniques (besides *retest-all*) required much less total effort than the safe technique at the cost of delaying fault detection for a few testing sessions.

**Minimization.** *Minimization* presents an interesting alternative. It had the smallest total effort (≈ 6% on the average), but had the highest fault age (11 regression test sessions on the average). Therefore it might be cost-effective when testing sessions must be short and when the cost of failures is low. We should also note that with *minimization* can't guarantee the maximum number of test cases selected.

**Random(n)**. The *random(n)* technique is arguably dominated by the *safe-random(n)* and *lru(n)* methods. Its average fault age is higher than theirs and has a much larger standard deviation. Also, its total effort is considerably higher than theirs.

**Safe-random(n).** On the average, this technique detected faults earlier and with less effort than both *lru(n)* and *random(n)*. One drawback is that the standard deviation of fault age is substantially higher than that of *lru(n)*. This appears to be because *safe-random(n)*, like *random(n),* allowed some faults to go entirely undetected. The effect of this in practice needs to be studied further. This technique might be cost-effective when the cost of failures is not too high.

**Lru(n).** This approach appeared to be competitive with others in terms of fault age and total effort, but was not the best performer under either measure. It did, however, have the lowest standard deviation for fault age and was the only technique that detected all faults before the experimental cutoff of 50 testing sessions. We believe that this property is quite interesting and deserves further study.

## 6.2 Model 2

In Model-2 $P_0$ contains all existing faults. After each regression

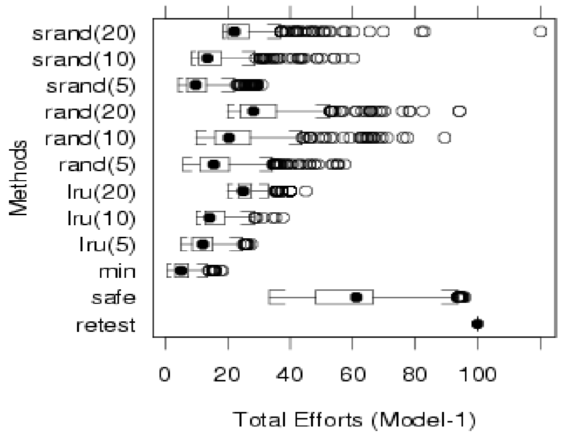

**Figure-2: Distribution of Total Effort (Model-1).**

testing session any newly identified faults are removed and regression testing is done once more. This continues until all faults are detected or 50 testing sessions have been conducted. In general we see that many faults are detected early in the process (after 1 or 2 testing sessions), but that the remaining faults are sometimes quite persistent (many runs were stopped only when they hit the cutoff of 50 testing sessions).

### 6.2.1 *Fault Age*

Figure-3 contains a box plot showing the distribution of fault age by RTS technique under Model-2. Table-4 shows the median, average, and standard deviation of fault age for each RTS technique.

All techniques had a low median fault age, but a substantially higher average fault age with a large standard deviation. This is because the majority of faults were detected during the initial testing sessions, while the remaining ones were sometimes quite hard to detect. Even retest-all and the safe technique were not able to immediately detect every fault (i.e., unlike in Model-1, some faults have age greater than 1).

Although for *minimization* the median fault age was 2, the average was roughly 6 with a standard deviation of about 12. Again, this is due to the fact many testing runs reached the 50-session cutoff without detecting all the faults.

We had expected that *lru(5)* would need no more than 20 or so testing sessions to detect all defects since that would have been enough to execute each test case at least once. But as shown by the outliers in Figure-3 some faults took over 30 test sessions to detect. Nevertheless, no *lru(n)* technique reached the cutoff of 50. Except for *random(20)* all other techniques did reach the cutoff point without detecting all faults.

We now compare the prioritization methods in more detail. The median fault ages are nearly the same, so we will focus on the average and standard deviation.

**LRU vs. Random**: For equal values of *n, lru(n)* has a slightly lower average fault age than *random(n)* does. Its standard deviation is also lower. An important difference between them,

**Table-4: Median, Average and Standard Deviation Fault Age by RTS Technique (Model-2).**

| | median | average | std. dev. |
|---|---|---|---|
| safe | 1 | 1.2 | 0.4 |
| retest-all | 1 | 1.2 | 0.4 |
| min | 2 | 6.1 | 11.8 |
| lru(5) | 1 | 4.5 | 5.6 |
| lru(10) | 1 | 2.7 | 2.8 |
| lru(20) | 1 | 1.8 | 1.5 |
| rand(5) | 2 | 5.9 | 9.0 |
| rand(10) | 1 | 3.5 | 5.3 |
| rand(20) | 1 | 2.1 | 2.5 |
| safe-rand(5) | 2 | 4.7 | 7.1 |
| safe-rand(10) | 1 | 2.7 | 3.8 |
| safe-rand(20) | 1 | 1.8 | 1.8 |

however, is that *lru(n)* detected every fault, while *random(n)* did not (i.e., *random(5)* and *random(10)* sometimes ran to the cutoff point of 50 testing sessions).

**Random vs. Safe-Random**: For equal values of *n, safe-random(n)* performed better than *random(n)* in terms of average fault age. Its standard deviation is also smaller.

**LRU vs. Safe-Random**: For equal values of *n,* the average fault age of *lru(n)* is nearly identical to that of *safe-random(n)*. The standard deviation of *lru(n)* is smaller, however.

### *6.2.2 Total Effort*

Figure-4 contains a box plot showing the distribution of total effort for each RTS technique. Table-5 shows the median, average, and standard deviation.

**NOTE:** Our measure of total effort under Model-2 is different from that of Model-1. Here we normalize by the size of the original test suite, not by the total effort expended by *retest-all.* We did this because behavior of *retest-all* varies considerably from program to program.

The total effort required by the *safe* technique (117% of original test suite size) is less than that required by *retest-all* (200% of original test suite size).

Here again, *minimization* required the least total effort (median value 51.7% of original test suite size or about 25% of effort required for *retest-all*). Yet, in contrast to Model-1, here *random(n)* required more effort than the *safe* method.

We now compare the prioritization methods in detail.

**LRU vs. Random**: For equal values of *n, lru(n)* requires much less effort than *random(n)* in terms of the median and average. In addition, the variance of *lru(n)* is much smaller.

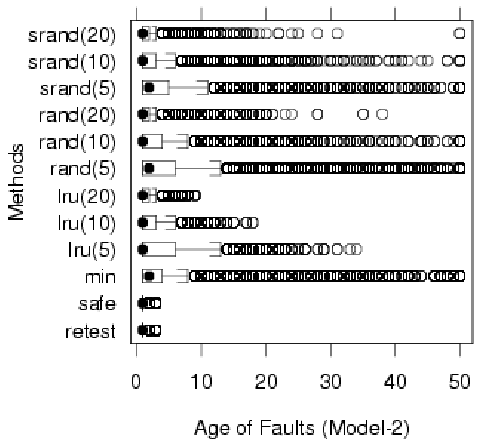

**Figure-3: Distribution of Fault Age (Model-2).**

**Random vs. Safe-Random**: For equal values of *n, safe-random(n)* requires less effort than *random(n)* in terms of the median and the average.

**LRU vs. Safe-Random**: For equal values of *n*, *safe-random(n)* performed better than *lru(n)* in terms of median, but there is no difference in the average total effort. On the other hand, we see that the variance is much higher for *safe-random(n*).

### *6.2.3 Cost-Benefit Tradeoffs*

When we consider both the costs and the benefits of the different RTS techniques for Model-2, we find both similarities and differences with Model-1.

**Safe**. As with Model-1 we assume that the safe technique cannot be used in a constrained environment. Again, the data is consistent with the assumption even though this specific point at which the technique exceeds its constraints will be situation-dependent. One difference is that under Model-2 the safe technique required less effort than the *random(n)* methods did.

**Minimization**. The minimization expended the smallest total effort and had the greatest average fault age. In contrast with Model-1, however, the difference in total effort between minimization and other techniques is less pronounced. This technique might be cost-effective when the cost of executing test cases is very high.

**Random(n).** As with Model-1, *random(n),* appears to be dominated by *safe-random(n)* and *lru(n)*. Under this model, however, the total effort is substantially higher than that of the other methods.

**Safe-random(n).** S*afe-random(n)* detects fault earlier and with less effort than *random(n)*, but behaves similarly to *lru(n)*. One limitation is that allowed faults to go undetected. This technique might be cost-effective when the cost of test execution is high and the cost of failures is low.

**Lru(n).** From the perspective of fault age and total effort, *lru(n)* may be more attractive under Model-2 than under Model-1.

**Table-5: Median and Average Total Efforts by RTS Technique (Model-2) as a percentage of the Original Test Suite Size**

| | median | average |
|---|---|---|
| safe | 117.2 | 119.6 |
| retest-all | 200 | 165.8 |
| min | 51.7 | 56.4 |
| lru(5) | 95.4 | 85.0 |
| lru(10) | 98.5 | 84.2 |
| lru(20) | 100 | 92.7 |
| rand(5) | 132.1 | 145.3 |
| rand(10) | 132.3 | 156.1 |
| rand(20) | 123.9 | 149.4 |
| safe-rand(5) | 62.6 | 83.3 |
| safe-rand(10) | 59.0 | 88.6 |
| safe-rand(20) | 77.1 | 96.0 |

Nevertheless, in both cases it detected all faults within the cutoff of 50 testing sessions. With the exception of *rand(20)*, no other prioritization techniques did that. Thus, *lru(n)* might be most especially cost-effective when the cost of failures is high.

## 7. CONCLUDING REMARKS

We have presented the initial results of an empirical study on using historical test execution data to prioritize test case selection in a constrained regression testing process. We investigated some of the costs and benefits of several RTS techniques under two different software evolution models. Our results highlight several differences among RTS and test case prioritization techniques, illustrate tradeoffs, and provide directions for further research.

As we discussed earlier, this study has several limits to its validity. Particularly, several threats to external validity limit our ability to generalize our results. These threats can only be addressed by extensive experiments with a wider variety of programs, test suites, series of versions, type of faults, etc. Keeping this in mind, we tentatively draw several conclusions.

Our experimental results strongly support our first conjecture that regression testing may have to done differently in constrained environments than non-constrained ones. They also support our second conjecture – that historical information may be useful in reducing costs and increasing the effectiveness of long-running regression testing processes.

As has been shown in other studies, *safe* techniques select widely varying and sometimes large numbers of test cases. In a constrained environment, such an approach may be simply infeasible. In other environments, of course, it may be a very powerful tool. Clearly, the decision to use or forego this technique must be made on a case-by-case basis. One interesting observation, however, was that under Model-2, the total effort for the safe technique was less than that of the *random(n)* method.

Minimization chose the smallest test suites, but was the weakest at

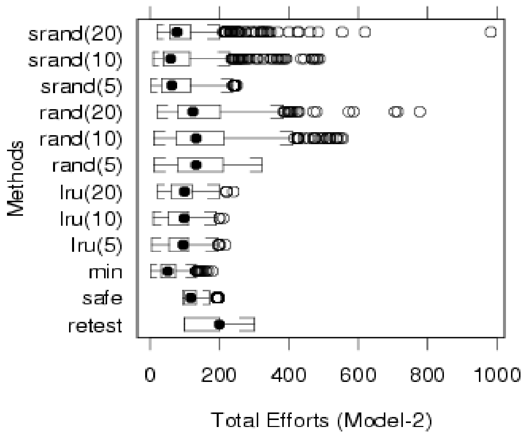

**Figure-4: Distribution of Total Efforts for Model-2**

detecting all faults. Nevertheless, it did detect many of the faults at low cost. For example, under Model-2 it detected most faults in one or two sessions while running only a handful of test cases. Thus, although it will miss some faults, it may be cost-effective for some part of the regression testing process (possibly in conjunction with some other technique). We should note however, that *minimization*, like *safe* and *safe-random(n)*, have substantial analysis costs that were not considered in this study.

Experience tells us that random techniques are cheap and reasonably effective, but it also tells us that their effectiveness increases considerably as n increases. This study, however, suggests that for severely constrained environments (i.e., at low settings of n) other approaches may be more attractive.

Under certain conditions *safe* methods guarantee that the selected test suite detect any defects that retest-all would have. As we see in this study, prioritization nullifies this guarantee. In fact, we saw that *safe-random(n)* had fault ages greater than 1 and that some faults escaped detection completely. Still its average fault age and total effort were better than those of other methods under Model-1. Under Model-2, only *lru(n)* did as well. As with *minimization*, analysis costs are high for *safe-random(n)*, but have not been factored into this study.

In terms of both fault age and total effort, *lru(n)* was competitive with other prioritization methods. One particularly interesting result was that the standard deviation of fault age when using *lru(n)* was less than that obtained from other methods. This appears to be because *lru(n)* always detected all faults long before the cutoff point. We assume that this is because *lru(n)* effectively cycles through the test suite eventually using each test case. We're obviously intrigued by this result and believe that it supports our conjecture that historical information may be useful in test case prioritization.

We are continuing this family of experiments. We plan to (1) improve our cost models to account for factors such as the overhead of each individual testing session and source code analysis costs, (2) extend our experiment to larger programs with a wider variety of naturally-occurring faults, (3) implement and evaluate other history-based prioritization techniques such as those described in Section 4, and (4) compare these methods to other non-history-based methods described in the literature.

## 8. ACKNOWLEDGEMENTS

This work was supported in part by grants from National Science Foundation Award CCR-0098158 to University of Maryland, and by National Science Foundation Award CCR-9707792 to University of Maryland. Siemens Laboratories supplied some of the subject programs. Alberto Pasquini, Phyllis Frankl, and Filip Vokolos provided the Space program and many of its test cases.

## 9. 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] R.G. Brown, Statistical Forecasting for Inventory Control. New York: McGraw-Hill, 1959.

[3] 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.

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

[5] T. Graves, M.J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *Proc. of the 20th Int'l. Conf. on Softw. Eng.,* pages 188-197, Apr. 1998.

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

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

[8] 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.*, pages191-200, May 1994.

[9] J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test application frequency. In *Proc. of the 22nd Int'l. Conf. on Softw. Eng.,* pages 126-135, Jun. 2000.

[10] H.K.N. Leung and L.J. White. Insights into regression testing. In *Proc. of Int'l. Conf. on Softw. Maint.*, pages 60-69, Oct. 1989.

[11] H.K.N. Leung and L.J. White. A cost model to compare regression test strategies. In *Proc. of Int'l. Conf. on Softw. Maint.*, pages 201-208, Oct. 1991.

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

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

[14] 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.

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

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

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

[18] 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, The Ohio State University, Mar. 1997.

[19] G. Rothermel and M.J. Harrold. Empirical studies of a safe regression test selection technique, *IEEE Trans. on Softw. Eng*., 25(6), pages 401-419, June 1998.

[20] G. Rothermel, R. Untch, C. Chu and M.J. Harrold. Test case prioritization: an empirical study. In *Proc. of Int'l. Conf. on Softw. Maint.*, pages 179-188, Aug. 1999.

[21] F.I. Vokolos and P.G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Proc. of the Int'l. Conf. on Softw. Maint.*, pages 44-53, Nov. 1998.

[22] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proc. of the 8th Int'l. Symp. on Softw. Rel. Engr.,* pages 230-238, Nov. 1997.