An Empirical Study of Regression Test Selection Techniques

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

April 28, 2000

Abstract

Regression testing is the process of validating modified software to detect whether new errors have been introduced into previously tested code, and provide confidence that modifications are correct. Since regression testing is an expensive process, researchers have proposed regression test selection techniques as a way to reduce some of this expense. These techniques attempt to reduce costs by selecting and running only a subset of the test cases in a program's existing test suite. Although there have been some analytical and empirical evaluations of individual techniques, to our knowledge only one comparative study, focusing on one aspect of two of these techniques, has been reported in the literature. We conducted an experiment to examine the relative costs and benefits of several regression test selection techniques. The experiment examined five techniques for reusing test cases, focusing on their relative abilities to reduce regression testing effort and uncover faults in modified programs. Our results highlight several differences between the techniques, and expose essential tradeoffs that should be considered when choosing a technique for practical application.

1 INTRODUCTION

As developers maintain a software system, they periodically *regression test* it, hoping to find errors caused by their changes, and provide confidence that their modifications are correct. To support this process, developers often create an initial test suite, and then reuse it for regression testing.

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

This tradeoff between the time required to select and run test cases and the fault detection ability of the test cases that are run is central to regression test selection. Because there are many ways in which to approach this tradeoff, a variety of test selection techniques have been proposed (e.g., [1, 4, 7, 8, 12, 15, 20]). Although there have been some analytical and empirical evaluations of individual techniques [4, 18, 20, 21], to our knowledge only one comparative study, focusing on one aspect of two of these techniques, has been reported in the literature [16].

We hypothesize that different regression test selection techniques create different tradeoffs between the costs of selecting and executing test cases, and the need to achieve sufficient fault detection ability. Because there have been few controlled experiments to quantify these tradeoffs, we conducted such a

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

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

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

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

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

study. Our results indicate that the choice of regression test selection algorithm significantly affects the cost-effectiveness of regression testing. Below we review the relevant literature, describe the test selection techniques we examined, and present our experimental design, analysis, and conclusions.

2 REGRESSION TESTING SUMMARY AND LITERATURE REVIEW

2.1 Regression Testing

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

1. Select $T' \subseteq T$, a set of 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 execution profile for P', from T, T', and T''.

Each of these steps involves important problems. Step 1 involves the *regression test selection problem*: the problem of selecting a subset T' of T with which to test P'. Step 3 addresses the *coverage identification problem*: the problem of identifying portions of P' or its specification that require additional testing. Steps 2 and 4 address the *test suite execution problem*: the problem of efficiently executing test suites and checking test results for correctness. Step 5 addresses the *test suite maintenance problem*: the problem of updating and storing test information. Although each of these problems is significant, we restrict our attention to the regression test selection problem.

Note that regression test selection is applicable both in cases where the specifications have not changed, and where they have changed. In the latter case, it is necessary to identify the test cases in T that are *obsolete* for P' prior to performing test selection. (Test case t is obsolete for program P' if and only if t specifies an input to P' that is invalid for P', or t specifies an invalid input-output relation for P'.) Having identified these test cases and removed them from T, regression test selection can be performed on the remaining test cases. Note further that the identification of obsolete test cases is necessary if any test case reuse is desired (whether by test selection or retest-all), because if we cannot effectively determine test obsolescence, we cannot effectively judge test correctness.

2.2 Regression Test Selection Techniques

A variety of regression test selection techniques have been described in the research literature. A survey by Rothermel and Harrold [19] describes several families of techniques; we consider three such families, along with two additional approaches often used in practice. We here describe these families and approaches, and provide a representative example of each; Rothermel and Harrold [19] and the references for the cited techniques themselves provide additional details. Later, in Section 3.2.3, we provide details on the specific techniques that we use in our experiments.

2.2.1 Minimization Techniques

Minimization-based regression test selection techniques (e.g., [5, 8]), hereafter referred to as *minimization techniques*, attempt to select minimal sets of test cases from T, that yield coverage of modified or affected portions of P.

For example, the technique of Fischer et al. [5] uses systems of linear equations to express relationships between test cases and basic blocks (single-entry, single-exit sequences of statements in a procedure). The technique uses a 0-1 integer programming algorithm to identify a subset T' of T that ensures that every segment that is statically reachable from a modified segment is exercised by at least one test case in T' that also exercises the modified segment.

2.2.2 Dataflow Techniques

Dataflow-coverage-based regression test selection techniques (e.g., [7, 15, 22]), hereafter referred to as *dataflow techniques*, select test cases that exercise data interactions that have been affected by modifications.

For example, the technique of Harrold and Soffa [7] requires that every definition-use pair that is deleted from P, new in P', or modified for P' be tested. The technique selects every test case in T that, when executed on P, exercised deleted or modified definition-use pairs, or executed a statement containing a modified predicate.

2.2.3 Safe Techniques

Most regression test selection techniques — minimization and dataflow techniques among them — are not designed to be *safe*. Techniques that are not safe can fail to select a test case that would have revealed a fault in the modified program. In contrast, when an explicit set of safety conditions can be satisfied, safe regression test selection techniques guarantee that the selected subset, T', contains all test cases in the original test suite T that can reveal faults in P'.

Several safe regression test selection techniques have been proposed (e.g., [4, 11, 20, 23]); the theory behind safe test selection and the set of conditions required for safety have been detailed in Rothermel and Harrold [19]. For example, the technique of Rothermel and Harrold [20] uses control-flow-graph representations of P and P', and test execution profiles gathered on P, to select every test case in T that, when executed on P, exercised at least one statement that has been deleted from P, or at least one statement that is new in or modified for P'.

2.2.4 Ad Hoc / Random Techniques

When time constraints prohibit the use of a retest-all approach, but no test selection tool is available, developers often select test cases based on "hunches", or loose associations of test cases with functionality. One simple approach is to randomly select a predetermined number of test cases from T.

2.2.5 Retest-All Technique

The retest-all technique simply reuses all existing test cases. To test P', the technique effectively "selects" all test cases in T.

2.3 Previous Empirical Work

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

Rosenblum and Weyuker [17, 18] and Rothermel and Harrold [20, 21] have conducted empirical studies to investigate whether certain regression test selection techniques are cost-effective relative to retest all.

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

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

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

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

2.4 Open Questions

None of the studies just described examined non-safe techniques, and none compared more than two techniques. Because non-safe techniques can discard *fault-revealing* test cases in T (test cases in T that would reveal faults in the modified program), whereas safe techniques, provided certain conditions are met, do not discard such test cases, the tradeoffs between test selection and fault detection should be explored, and these techniques should be compared.

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

- How do techniques differ in terms of their ability to reduce regression testing costs?
- How do techniques differ in terms of their ability to detect faults?
- What tradeoffs exist between test suite size reduction and fault detection ability?
- When is one technique more cost-effective than another?
- How do factors such as program design, location and type of modifications, and test suite design affect the efficiency and effectiveness of test selection techniques?

It is these questions that we wish to address through our empirical studies.

3 THE EXPERIMENT

3.1 Operational Model

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

3.1.1 Modeling Cost

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

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

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

Given this model, we needed to measure two things: the reduction in the cost of executing and validating test cases, and the average analysis cost. Given our assumptions, the former can be measured in terms of test suite size reduction, as $(\frac{|T'|}{|T|})$. For several reasons, however, we did not measure analysis costs directly. Most important, we did not possess implementations of all techniques, and instead were required to simulate techniques for which we had no implementations. Furthermore, because the experimental design required us to run over 264,400 test suites, we used several machines. We did not believe that the performance metrics taken from different machines were comparable. Instead we try to estimate how large analysis costs can be before they outweigh reductions in test suite size (see Section 4.3).

3.1.2 Modeling Fault Detection Effectiveness

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

On a per-test-case basis: One way to measure a reduction in the fault detection effectiveness of a regression test selection technique, given program P and faulty version P', is to identify those test

cases that are in T and reveal at least one fault in P', but that are not in T'. This quantity can then be normalized by the number of fault-revealing test cases in T. One problem with this approach is that multiple test cases may reveal a given fault. In this case some test cases could be discarded without reducing fault detection effectiveness; however, this measure penalizes such a decision.

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

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

It is important to note that both of these approaches measure a test suite's ability to detect at least one fault. They do not measure the exact number of faults detected. As we shall discuss further below, this distinction is unimportant for all but one of our subject programs, as they have versions that each contain exactly one fault.

3.2 Experimental Instrumentation

Program Name	*Number of Functions*	*Lines of Code*	*Number of Versions*	*Test Pool Size*	*Average Test Suite Size*
replace	21	516	32	5542	398
print_tokens	18	402	7	4130	318
print_tokens2	19	483	10	4115	389
schedule2	16	297	10	2710	234
schedule	18	299	9	2650	225
totinfo	7	346	23	1054	199
tcas	9	138	41	1608	83
space	136	6218	33	13585	4361
player	766	49316	5	1033	154

Table 1: Subjects.

3.2.1 Programs

For our study, we used nine C programs, with a number of modified versions and test suites for each program. The subjects come from three sources:

- a group of seven C programs collected and constructed initially by Hutchins et al. [9] for use in experiments with dataflow- and control-flow-based test adequacy criteria,
- an interpreter for an array definition language, used within a large aerospace application, `space`,
- one large subsystem, `player`, from the internet game `Empire`.

We slightly modified some of the programs and versions in order to use them with our tools. Table 1 describes these subjects, showing the number of functions, lines of (non-comment) code, distinct versions, test pool size, and the size of the average test suite. We describe these and other data in the following paragraphs.

The Siemens Programs. Seven of our subject programs originated with a previous experiment performed by Hutchins et al. [9]. 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. constructed a *test pool* of black-box test cases [9] using the category partition method and Siemens Test Specification Language tool [2, 14]. They then added additional 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 (between 7 and 41 versions) 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. Next, they discarded modifications that they considered either

too easy to find (found by more than 350 test cases) or too difficult to find (found by fewer than three) with their previously developed test cases.

Space. `Space` has been used as a subject for several empirical studies of testing [23, 24, 25]. As Table 1 indicates, it contains 136 C functions and 6,218 lines of (non-comment) code. The program functions as an interpreter for an array definition language (ADL): it reads a file that contains several 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 a data file containing a list of array elements, positions, and excitations; otherwise the program outputs error messages.

`Space` has 33 versions, each containing a single fault that was discovered during the program's development or later by the authors of this paper.

The test pool for `space` was constructed in two stages. First we obtained a pool of 10,000 test cases from Vokolos and Frankl, who had constructed the pool for another study by randomly generating test cases [24]. We then added new test cases until every executable edge in the control flow graph was exercised by at least 30 test cases.[1] This process yielded a test pool of 13,585 test cases.

Player. `Player` is the largest subsystem of the internet game `Empire`. As Table 1 indicates, it contains 766 functions (all written in C) and 49,316 lines of (non-comment) code. `Player` is essentially a transaction manager that operates as a server. Its main routine consists of initialization code followed by a five-statement event loop in which execution pauses and waits for receipt of a user command. Users communicate with the server by running a small client program that takes user input and passes it as commands to `player`. When `player` receives a command it processes the command — usually by invoking one or more subroutines — and then waits for another command. While processing commands, `player` may return data to the user's client program for display on the user's terminal, or write data to a local database (a directory of ASCII and binary files) that keeps track of game state. The event loop and the program terminate when a user issues a "quit" command.

Since its creation in 1986, the `Empire` code has been enhanced and corrected many times, with most changes involving the `player` subsystem. For this experiment we located a "base" version of `player` with five distinct modified versions (see Table 2). Each version had been created by merging, multiple, often unrelated, changes made by one or more independent coders. These versions, therefore, do not form a sequence of modifications of the base program; rather, each is a unique modified version of the base version.

Version	*Functions Modified*	*Lines of Code Changed*
1	3	114
2	2	55
3	11	726
4	11	62
5	42	221

Table 2: Modified versions of `player`.

`Player` is an interesting subject for several reasons. First, the program is part of an existing software system that has a long history of maintenance at the hands of numerous coders, and in this respect, the system is similar to many existing commercial software systems. Second, as a transaction manager, `player` is representative of a large class of software systems that receive and process interactive user commands, such as database management systems, operating systems, menu-driven systems, and computer-aided drafting systems. Third, we were able to locate real modified versions of one base version of `player`. Finally, although not huge, the program is not trivial.

There were no test cases available for `player`. Therefore, we created our own test cases using the `Empire` information files as an informal specification. These files describe the 154 `player` commands and describe the parameters and special effects associated with each.

The test cases we constructed exercise each parameter, special effect, and erroneous condition described in the information files. Because the complexity, parameters, and effects of commands vary widely, we had to create between one and 30 test cases for each, ultimately producing a test pool of 1033 test cases. To avoid a possible source of bias, we constructed this test pool prior to examining the code of the modified versions.

[1] We treated 17 edges exercisable only on `malloc` failures as nonexecutable.

Each test case is a sequence of between one and 28 lines of ASCII text representing potential user commands. To use these test cases, however, some additional scaffolding was needed. Therefore we created a testing script and several accompanying tools.

Note that, unlike the Siemens programs and `space`, the modified versions of `player` do involve specification changes. In our test case creation, however, we ensured that no test cases in the test pool were obsolete for any of the `player` versions. (Section 2.1 describes the necessity of this.) Thus, regression test selection can be applied to all of these test cases, on each of the versions.

3.2.2 Tests, Test Pools, Versions, and Test Suites

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

To create test suites for `player` we used a different approach. We viewed each `player` command as a unit of functionality, and created function-coverage-adequate test suites by selecting, from the set of test cases for each command, one test case. We generated 100 such test suites, each containing 154 test cases.

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

These test cases differ in their ability to detect faults. Figure 1 uses boxplots[2] to depict the distribution of the proportion of fault-revealing test cases for all test suites used in our studies over all programs. We see that the effectiveness of these test suites differs substantially across different programs.

Test cases for the Siemens programs find known faults with probability between .06% and 19.77%, while those for `space` find faults with probability between .04% and 94.35%. While the range is wide for `space`, the percentage of fault detecting test cases range from .77% to 4.55% for `player`. Over all versions, the median percentage of test cases detecting a fault is less than 7%.

3.2.3 Test Selection Techniques

To perform the experiments, we required implementations or simulations of several regression test selection techniques. Note that in all cases in which simulation was necessary, our simulations were designed to ensure that we could obtain exact results with respect to test cases selected, allowing exact test suite size reduction measures, and exact fault detection effectiveness measures.

Minimization Technique. As a *minimization* technique, we created a simulator tool that selects a minimal test suite T', such that T' is edge-coverage-adequate for a set of edges, in the control flow graphs for P or P', that are associated with code modifications. In this context, we considered an edge to be associated with a code modification if its sink is a node corresponding to a statement that has been deleted or modified for P', or added to P'.

To perform this process on the Siemens programs and `space`, we used Rothermel and Harrold's regression test selection tool `DejaVu` (described below in reference to our safe technique), which selects exactly the desired edges in cases (such as these) where programs do not contain multiple modifications. We then used test case execution information obtained through profiling to determine the test cases associated with each edge, and we selected one test case through each such edge.

[2] In a boxplot, a box represents a data distribution. The box's height spans the central 50% of the data and its upper and lower ends mark the upper and lower quartiles. The bold dot within the box denotes the median. The T-shaped whiskers indicate the 10% and 90% quantiles, respectively. All other detached points are "outliers". For comparing the averages of data, the median is more appropriate than the mean since it is less influenced by outliers.

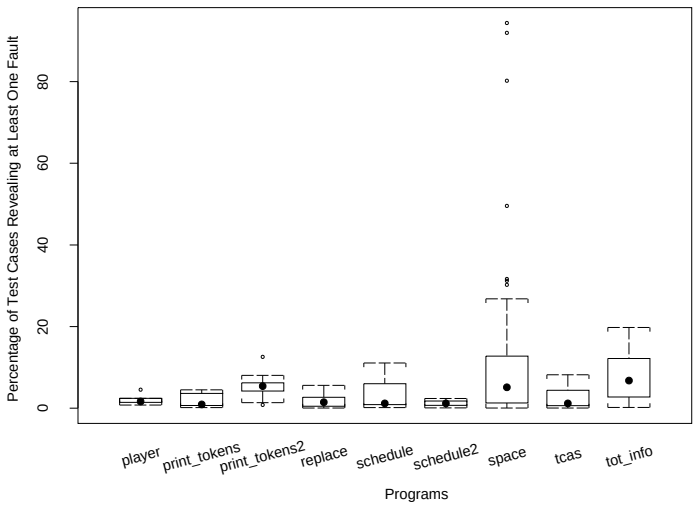

Figure 1: Percentage of fault-revealing test cases across all program versions.

On `player` we required a different approach due to its inclusion of multiple modifications. We hand-instrumented the code of `player`, at the entry to each modification point (at each location corresponding to an edge entering that point), to report the set of test cases that reached that point. We then selected one test case through each point for inclusion in T', and when all selection was complete we eliminated duplicate test cases.

Dataflow Technique. As a *dataflow* technique, we simulated a tool by manually inspecting program modifications, and generating a list of tuples that represent the definition-use pairs that are affected by the modifications. In this context, we considered a definition-use pair to be affected by a modification if it had been deleted from P in obtaining P', or if it involved a definition or use contained in a statement or predicate that had been modified in creating P'.

We used a dataflow testing tool [6] to identify the test cases in the test suite for each program that satisfied the affected definition-use pairs for each version of that program. For each version, we created a set of selected test cases T' that contained all such test cases.

The first step of this simulation process was human-labor-intensive, and it was not feasible to perform this process on the `space` and `player` programs; thus, we were able to apply this technique only to the Siemens programs.

Safe Technique. As a *safe* technique, we used an implementation of Rothermel and Harrold's regression test selection algorithm, implemented as a tool called `DejaVu`, and integrated with the `Aristotle` program analysis system. [3] We provide a brief overview of the approach here; Rothermel and Harrold [20] presents the underlying test selection algorithm and tool in detail, with cost analyses and examples of its use.

`DejaVu` constructs control-flow graph representations of the procedures in two program versions P and P', in which individual nodes are labeled by their corresponding statements. The approach assumes that a *test history* is available that records, for each test case t in T and each edge e in the control flow graph for P, whether t traversed e. This test history is gathered by instrumentation code that is inserted into the source code of the system under test.

`DejaVu` performs a simultaneous depth-first graph walk on a pair of control flow graphs for each procedure and its modified version in P and P', keeping pointers to the current node reached in each graph. During this walk, the algorithm examines the statements associated with the nodes in the two

[3] Our experiments used the original tool on the Siemens programs and `space`; on `player`, it was necessary to simulate a portion of the tool's operations; however, the use of this simulation affects only tool analysis time, not test selection, and thus does not impact our results.

graphs, and the edges (representing control flow) leaving those nodes. When these are identical, the algorithm continues its walk at the successor nodes; when these are non-identical, the algorithm places the edge it just followed into a set of "dangerous edges" and returns to the source of that edge, ending that trail of recursion. After the algorithm has determined all the dangerous edges that it can reach by crossing non-dangerous edges, it terminates. At this point, any test case $t \in T$ is selected for retesting P' if the execution trace for t – a listing of the edges, in the control flow graph for P, that were traversed by t when it was executed previously on P – contains a dangerous edge.

`DejaVu` guarantees safety as long as equivalent execution traces on P and P' for identical inputs imply that P and P' will produce equivalent behaviors. As long as the three assumptions discussed in Rothermel and Harrold [20] hold, this condition is met and `DejaVu` is safe, necessarily selecting at least all test cases in T that could, if executed on P', reveal faults in P'. (In brief, the three necessary assumptions are that (1) the test cases in T produced correct results when executed on P, (2) any test cases in T that are obsolete for P' (no longer represent specified input-output relations for P') have been removed from T, and (3) the testing environment can be controlled such that P and P' execute deterministically on T.) For the programs and test suites that we studied, these conditions were met.

Random Technique. As a random technique we created a tool that, given a selection percentage n and a test suite T, randomly selects $n\%$ of the test cases from T, outputting T', a reduced test suite containing only the selected test cases.

Retest-All Technique. The retest-all technique required no implementation.

3.3 Experimental Design

3.3.1 Variables

The experiment manipulated three independent variables:

1. the subject program (9 programs, each with many modified versions);
2. the test selection technique (safe, dataflow, minimization, random(25), random(50), random(75), retest all); and
3. test suite composition (coverage-based or random).

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

1. the ratio of the number of test cases in the selected test suite T' to the number of test cases in the original test suite T; and
2. whether one or more test cases in T' reveals at least one fault in P'.

For each combination of program, version, technique,[4] and test suite composition type we used 100 of the associated test suites. From these 100 data points we computed two dependent variables.

1. average reduction in test suite size, and
2. fault detection effectiveness (1 minus the percentage of test suites in which T would have revealed at least one fault in P', but T' did not).

3.3.2 Design

This experiment uses a full-factorial design with 100 repeated measures. That is, for each subject program, we selected 100 coverage-based and 100 random test suites from the test-suite universe. Then for each program version we applied each applicable test selection technique to each of the 200 test suites. Finally, we evaluated the fault detection effectiveness of the resulting test suites.

[4] Actually, for each applicable technique: recall from Section 3.2.3 that we did *not* apply the dataflow technique to `space` or to `player`.

3.3.3 Threats to Internal Validity

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

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

3.3.4 Threats to External Validity

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

Artifact representativeness is a threat when the subject programs are not representative of programs found in industrial practice. There are several such threats in this experiment. First, most of the subject programs (the Siemens programs) are of small size. As discussed earlier, there is some evidence to suggest that larger programs allow greater test suite size reduction, although at higher cost, than smaller programs do. Thus, larger programs may be subject to different cost-benefit tradeoffs. We have begun to address this problem by studying the larger `space` and `player` programs. As we collect other large programs with versions and test cases we will be able to further limit, but not eliminate, these problems. Also, in most of the programs (Siemens and `space`) there is exactly one fault in every subject program. Industrial programs have much more complex fault patterns. Again, we will have to obtain further experimental subjects and improve our measurement infrastructure in order to capture exactly which of several faults are discovered by a test suite. We have begun to explore such improved measurement techniques in our research [10].

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

3.3.5 Threats to Construct Validity

Threats to construct validity arise when measurement instruments do not adequately capture the concepts they are supposed to measure. For example, in this experiment our measures of cost and effectiveness are very coarse. For instance, they treat all faults as equally severe. Another problem is that our measure of fault detection effectiveness captures the ability of a test suite to identify at least one fault. Ideally, this measure should, instead, capture a test suites' ability to detect all faults in a system. For the Siemens programs and `space`, which have exactly one fault in each version, these definitions are equivalent. For `player`, our effectiveness measures may be inflated. As we discussed earlier, we are currently experimenting with new approaches to gathering this information.

3.3.6 Analysis Strategy

Our analysis strategy has three steps. First we summarize the data. Then we compare the ability of the test selection techniques to reduce test suite size, and we compare the fault detection effectiveness of the resulting test suites. Finally, we make several comparisons between program-analysis-based (i.e., minimization, safe, and dataflow) and random techniques. For example, in one analysis we explore how large analysis costs can become before the program-analysis-based techniques become less cost-effective than random ones.

4 DATA AND ANALYSIS

Two sets of data are important for this study: the test selection and the fault detection summaries. This information is captured for every test suite, every subject program, and every test selection technique. The test selection summary gives the size (in number of test cases) of T and T'. From this information we calculate the percentage reduction in test suite size. The fault detection summary shows whether

T and T' contain any fault-revealing test cases. From this information we determine whether the test selection technique compromised fault detection effectiveness.[5]

In addition to our use of boxplots to display data (as described in Section 3.2.2), we also use arrays of boxplots (a type of Trellis display [3]) to show data distributions that are conditioned on one or more other variables (e.g., Figure 2). By conditioned, we mean that data are partitioned into subsets, such that the data in each subset have the same value for the conditioning variable. For example, Figure 3 depicts the fault detection effectiveness for test suites created by different techniques, conditioned on the program on which the test suite was run. That means that the data is partitioned into nine subsets; one for each program. And then we draw one boxplot for each subset.

4.1 Test Suite Size Reduction

Figure 2 depicts the ability of each technique to reduce test suite size, conditioned on program. For these programs, we see that the random techniques extract a constant percentage of the test cases (by construction) and that minimization (by nature of the modifications made to the subjects) almost always (in 84% of the cases, most of the exceptions occuring for `player`) selects only 1 test case. The safe and the dataflow techniques behave similarly on the Siemens programs (median reduced suite size is 60% for coverage-based suites and 54% for random). Interestingly, the safe technique performs best on the two large programs: median reduced suite size is roughly 5% for `player` and 20% for `space`.

4.2 Fault Detection

Figure 3 depicts the fault detection effectiveness of test suites selected with each technique, conditioned on program. Overall, we found that minimization had the lowest fault detection effectiveness. The effectiveness of the random techniques increased with test suite size, but that the rate of increase diminished as size increased. Again the safe and dataflow techniques exhibited similar median performances on the Siemens programs, but the dataflow distribution has several outliers (e.g., for the `schedule`, `schedule2`, and `print_tokens2` programs). This occurs because in some cases the dataflow technique allows faults to go undetected, while the safe technique does not.

One interesting observation is that the fault detection effectiveness of test suites chosen by the minimization technique is particularly high for `player`. One reason for this is that versions of `player` contain multiple modifications. Thus, the average size of the selected test suite is larger for `player` than for other programs, giving these test suites more chances to identify faults.

4.3 Cost-Benefit Tradeoffs

Figure 4 depicts tradeoffs between test suite size reduction versus fault detection effectiveness of each selection technique. Each panel in Figure 4 is a scatterplot depicting the performances of one regression test selection technique. Each scatterplot contains a number of points and one "X" symbol. Each point represents the performance of the associated regression test selection technique when applied to a program-version and test suite pair. Each point is plotted at position (x,y), where x is the reduced test suite size and y is the fault detection effectiveness of the reduced test suite. The x-coordinate of the "X" symbol is equal to the median reduced test suite size for the observations depicted in the scatterplot. The y-coordinate is equal to the median fault detection effectiveness.

For random techniques the selected test suite size is predetermined, while its effectiveness is unknown in advance. Random techniques were very effective in general (median 88% for random25). Overall, as the selection ratio grows, effectiveness also tends to grow, but the rate of growth diminishes.

The safe technique always had 100% effectiveness, but its reduced test set sizes vary widely (from 0% to 100%). Dataflow shows very similar performance, but since it is not safe, it can fail to select some fault detecting test cases.

Minimization, on the other hand, chose very few test cases, while its effectiveness varied widely (0% to 100%).

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

In this section we explore the effect of analysis costs for non-random techniques on the relationships in Figure 4. To do this we examine how each non-random technique compares to random techniques

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

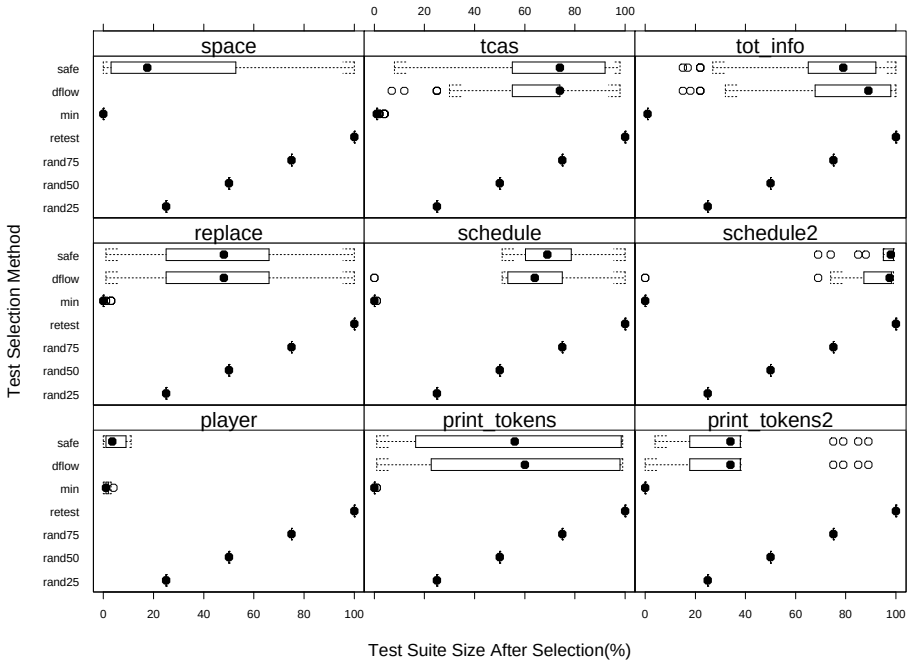

Figure 2: Test suite size reduction by selection technique, conditioned on program.

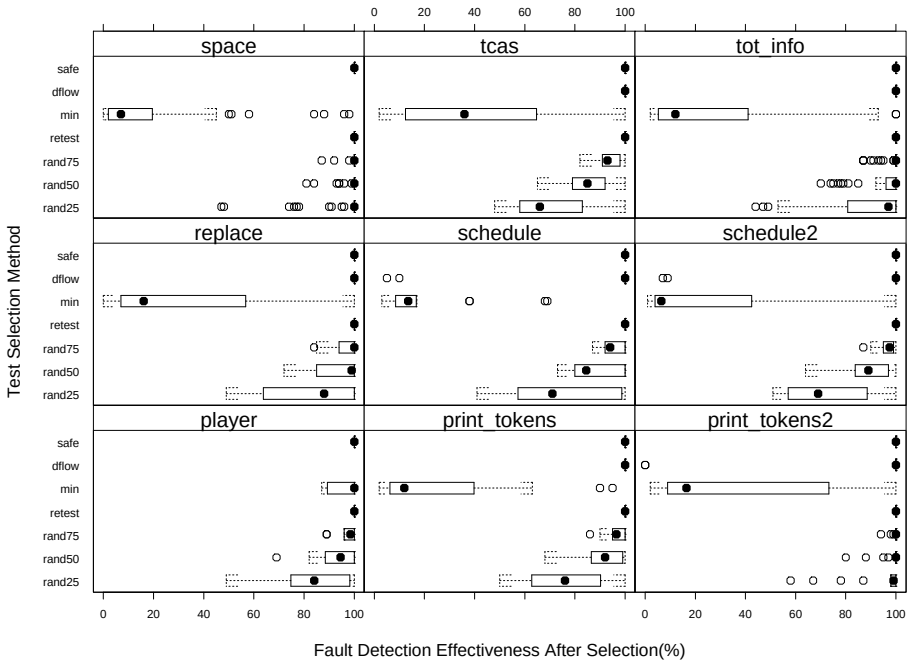

Figure 3: Fault detection effectiveness by selection technique, conditioned on program.

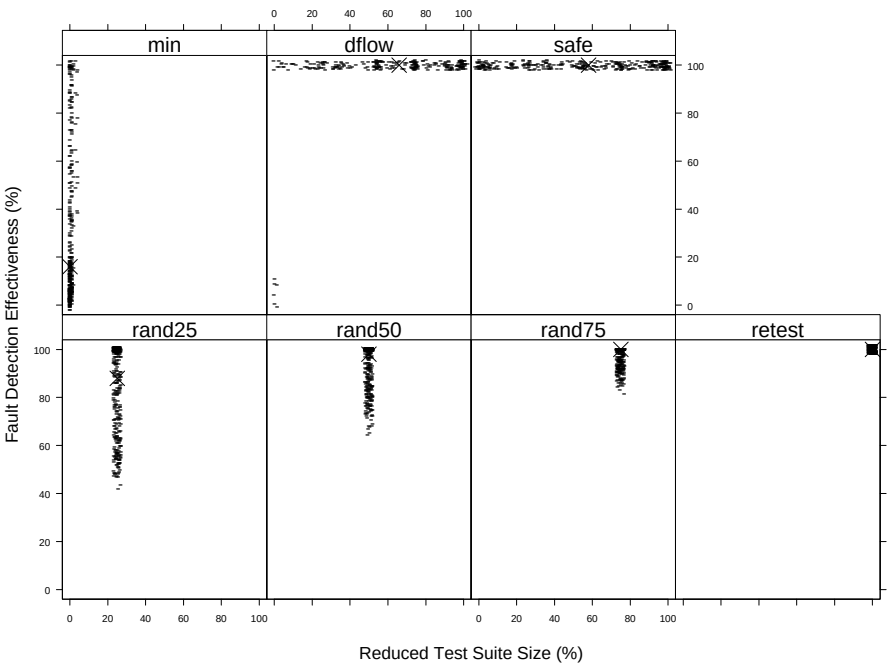

Figure 4: Fault detection effectiveness and test suite size, irrespective of analysis cost.

and to each other. We assume that the analysis costs for non-random techniques can be stated in terms of the cost to run a single test case (analysis costs for random techniques are nearly 0), and then we characterize how many test cases can be run (i.e., how long analysis can take) before the non-random technique becomes less cost-effective than random ones.

We begin with minimization, the rule with the smallest test suites and lowest fault detection effectiveness. We will compare its detection rate to that of a randomized rule calibrated to have the same total computational cost. Our goal is to find an upper bound, k, on the analysis cost of minimization. That is, if the analysis costs are greater than the cost of running k test cases, then there exists a random technique that is less expensive and has the same fault detection effectiveness.

We then perform similar analyses comparing the safe technique to other randomized rules and to retest all. Conceivably, we could perform a similar analysis comparing the dataflow technique to other randomized rules as well. In our experiment, however, the safe and dataflow techniques behaved similarly on the Siemens programs: their fault detection effectiveness results were nearly identical, and for many versions they chose the same average numbers of test cases per test suite. Moreover, we are not able to apply the dataflow technique to `space` or `player`. Consequently, in the next section we analyze only the safe technique.

After each comparison we discuss our interpretations and their limitations.

4.3.1 Minimization versus Randomization

The test suites selected by minimization were both the smallest and the least effective of all selected test suites. In 84% of the cases minimization chose exactly one test case and it never chose more than twelve.[6] On the median, minimization test suites found 16% of the faults that would have been found by retest-all.

Techniques other than minimization typically selected test suites with on the order of 100-200 test cases, and were much more effective at detecting faults. Because minimization does choose very few test cases, there may be situations in which its use is appropriate – namely when test cases are very expensive to run and missed faults are not considered excessively costly. Therefore, further study is warranted. In

[6] Of course, this is to be expected since most program versions contained exactly one change. For programs with multiple changes, larger minimization suites would be selected, as with `player`.

Coverage Suites ($k = 2.7$)									
	player	print_tokens	print_tokens2	replace	schedule	schedule2	space	tcas	tot_info
Random	0	4	1	12	7	9	3	36	13
Min	5	3	9	20	2	1	30	5	10
Random Suites ($k = 4.65$)									
	player	print_tokens	print_tokens2	replace	schedule	schedule2	space	tcas	tot_info
Random	0	4	1	12	4	8	10	31	14
Min	5	3	9	20	5	2	23	10	9

Table 3: The number of program versions for which the modified Random and Minimization techniques outperformed each other.

particular, we are also interested in knowing how much analysis cost minimization can incur before a random technique would be preferable.

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

Ideally we would like to compare minimization to a rule that chooses $100p\%$ test cases at random, where this is equal to the average size of minimization test suites. In our experiment we constructed random test suites only with $p \in \{0.25, 0.5, 0.75, 1\}$ (and, in effect, $p = 0$). So we simulate the long-run behavior of an arbitrarily-sized random technique by randomizing over values of p for which we have test suites. For instance, if we want to simulate random(5), we use random(25) with probability 0.2, and do no regression testing at all (random(0)) with probability 0.8 (our experiments suggest that this approach underestimates the effectiveness of the true random technique, and, thus, overestimates the value of k that we are looking for).[7]

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

For coverage-based test suites, we found that for $k = 2.7$, the randomized rule had higher detection rates in 85 program-versions and minimization had higher detection rates in 85 program-versions. For random test suites, we found that for $k = 4.65$, the randomized rule had higher detection rates in 84 program-versions and minimization had higher detection rates in 86 program-versions (see Table 3).

These results suggest that, for the programs we studied, the analysis costs for minimization must be very small (less than the cost of running five test cases on the average) in order for minimization to be cost-effective.

4.3.2 Safe versus Randomization

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

We found that for coverage-based test suites there exists a randomized rule with the same average test suite size (i.e., $k = 0$) as the safe technique that finds faults 96.7% ($p = 0.033$) as often in half the program-versions as the safe technique does. When $k = 0.1$ there is a randomization rule as costly as the safe technique that detects faults 99% as often in half the program-versions.

For random test suites $p = 0.11$ when $k = 0$: a random rule with the same size test suites as safe finds 89% of the faults that safe did in half the program-versions. When $p = 0.05$, $k = 10$ and when $p = 0.01$, $k = 25$ (see Table 4).

These results suggest that, for the programs we studied, the analysis costs that the safe technique can incur before becoming cost-ineffective depend on the level of fault detection effectiveness we would accept from a randomly selected test suite. The higher the effectiveness, the more analysis costs we should be willing to incur.

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

Coverage Suites ($k = 0, p = 0.033$)									
	player	print_tokens	print_tokens2	replace	schedule	schedule2	space	tcas	tot_info
Random	0	5	6	13	5	9	15	10	22
Safe	5	2	4	19	4	1	18	31	1
Random Suites ($k = 0, p = 0.11$)									
	player	print_tokens	print_tokens2	replace	schedule	schedule2	space	tcas	tot_info
Random	0	5	6	13	4	9	15	12	22
Safe	5	2	4	19	5	1	18	29	1

Table 4: The number of program versions for which the modified Random and Safe techniques outperformed each other.

4.3.3 Safe versus Retest-all

The safe technique always found all faults that could be found given the test suites used. Therefore, a safe technique is preferable to running all test cases in the test suite if and only if analysis costs are less than the costs of running the unselected test cases. Figure 2 contains data showing the sizes of test suites selected by the safe technique. It demonstrates that test suite reduction depends dramatically on the program: selected test suites for `schedule2` were typically 99% as large as the original suites, while those for `player` are about 5% as large.

5 SUMMARY AND CONCLUSIONS

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

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

- Minimization produced the smallest and the least effective test suites. Although fault detection is obviously important, there are cases where testing is very expensive. In these cases minimization may be cost-effective. Nevertheless, for the programs and test suites we studied, random selection of just slightly larger suites (less than five more test cases) yielded fault detection results equivalent to those of minimization (on average) with little analysis costs. One limitation here is that "on the average" applies to long-run behavior. Half of the time the random technique was as effective as minimization, half of the time it was not. If greater confidence is required, then the random techniques will need to select more than five additional test cases. One approach to understanding this issue better would be to restructure the analyses of Section 4.3 to include a desired confidence level.

  Another limitation is that, in practice, we cannot know how many test cases minimization (or any other regression test selection algorithm) would pick without actually running it. One approach to tackling this issue might be found in developing prediction models for regression test selection techniques (e.g., as in Rosenblum and Weyuker [18]).

- The safe and dataflow techniques had nearly equivalent average behavior in terms of cost-effectiveness, typically detecting the same faults, and selecting the same size test suites. However, because dataflow techniques require at least as much analysis as the two most efficient safe techniques [4, 20], we saw little reason to recommend dataflow if test selection alone is the goal. However, dataflow techniques can be useful in other components of regression testing, such as in identification of portions of P' that are not adequately tested by T. In other words, our model does not capture all possible costs or benefits of regression test selection techniques, and thus, may be too coarse for some situations.

- The safe technique found all faults for which we had fault-revealing test cases while selecting 60% of the test cases on the median. However, we saw that for several programs it could not reduce the test suites at all. Also, we found that, on the average, only slightly larger random test suites could be nearly as effective. Again, we have to remember that we are making a probabilistic assessment. This raises an important measurement question. That is, when should we analyze techniques like these on a case by case basis, and when is an amortized analysis more appropriate. We are currently exploring alternate analysis techniques [10].

- We found that our results were sensitive not only to the regression test selection techniques we used, but also to the programs, the characteristics of the changes, and the composition of the test suites. We believe that it is important to understand more precisely how these factors affect our techniques. Without this information, we may mistake the effect of a non-representative workload for differences in techniques. This problem is related to the problem of developing prediction models for regression test selection techniques. It will also be important to examine a broader range of subject programs.

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

6 ACKNOWLEDGMENTS

This work was supported in part by grants from Microsoft, Inc. to Ohio State University and Oregon State University, by National Science Foundation National Young Investigator Award CCR-9696157 to Ohio State University, by National Science Foundation Faculty Early Career Development Award CCR-9501354 to University of Maryland, by National Science Foundation Faculty Early Career Development Award CCR-9703108 to Oregon State University, by National Science Foundation Award CCR-9707792 to Ohio State University, University of Maryland, and Oregon State University, by National Science Foundation Grant SBR-9529926 to the National Institute of Statistical Sciences, and by an Ohio State University Research Foundation Seed Grant. Siemens Laboratories supplied several of the subject programs. Alberto Pasquini, Phyllis Frankl, and Filip Vokolos provided `space` and many of its test cases. Chengyun Chu assisted with further preparation of the `space` program and development of its test cases. Rui Wu and Lei Cao collected some of the data for the dataflow testing experiments.

References

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

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

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

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

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

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

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

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

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

[10] 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.*, June 2000.

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

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

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

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

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

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

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

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

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

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

[21] G. Rothermel and M. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Trans. on Softw. Eng.*, 24(6):401–419, June 1998.

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

[23] F. Vokolos and P. Frankl. Pythia: A regression test selection tool based on textual differencing. In *Int'l. Conf. on Reliability, Quality and Safety of Softw. Intensive Sys.*, May 1997.

[24] 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.

[25] W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini. Test set size minimization and fault detection effectiveness: A case study in a space application. In *Proc. of the 21st Annual Int'l. Comp. Softw. & Appl. Conf.*, pages 522–528, Aug. 1997.