6-1998

Empirical Studies of a Safe Regression Test Selection Technique

Gregg Rothermel
University of Nebraska-Lincoln, gerother@ncsu.edu

Mary Jean Harrold
Ohio State University

Empirical Studies of a Safe Regression Test Selection Technique

Gregg Rothermel, *Member*, *IEEE*, and Mary Jean Harrold, *Member*, *IEEE*

Abstract—Regression testing is an expensive testing procedure utilized to validate modified software. Regression test selection techniques attempt to reduce the cost of regression testing by selecting a subset of a program's existing test suite. Safe regression test selection techniques select subsets that, under certain well-defined conditions, exclude no tests (from the original test suite) that if executed would reveal faults in the modified software. Many regression test selection techniques, including several safe techniques, have been proposed, but few have been subjected to empirical validation. This paper reports empirical studies on a particular safe regression test selection technique, in which the technique is compared to the alternative regression testing strategy of running all tests. The results indicate that safe regression test selection can be cost-effective, but that its costs and benefits vary widely based on a number of factors. In particular, test suite design can significantly affect the effectiveness of test selection, and coverage-based test suites may provide test selection results superior to those provided by test suites that are not coverage-based.

Index Terms—Software maintenance, regression testing, selective retest, regression test selection, empirical studies.

————————————— ✦ —————————————

1 INTRODUCTION

REGRESSION TESTING is an expensive testing process that attempts to validate modified software and ensure that new errors are not introduced into previously tested code. One method for reducing the cost of regression testing is to save the test suites that are developed for a product, and reuse them to revalidate the product after it is modified. One regression testing strategy reruns all such tests, but this *retest-all* approach may consume excessive time and resources. *Regression test selection techniques*, in contrast, attempt to reduce the cost of regression testing by selecting a subset of the test suite that was used during development and using that subset to test the modified program.

Over the past two decades, much effort has been expended on research into regression test selection techniques, and many techniques have been described in the literature [1], [3], [4], [5], [6], [7], [8], [9], [10], [12], [15], [16], [17], [18], [21], [22], [23], [24], [25], [28], [35], [36], [37], [38], [39], [40], [41]. These techniques have been evaluated and compared analytically [33], but only recently have attempts been made to evaluate or compare them empirically [6], [8], [11], [29], [30], [34], [40].

In previous work, we developed and described a new regression test selection technique [34]. We proved that under certain well-defined conditions, our test selection algorithms exclude no tests (from the original test suite) that if executed would reveal faults in the modified program. Under these conditions, our algorithms are *safe*, and their fault-detection abilities are equivalent to those of the retest-all

approach. Despite these analytical results, to draw conclusions about the ability of our algorithms to reduce regression testing costs and detect faults in practice, we require empirical data.

To investigate the costs and benefits of using our regression test selection technique, and the factors that may influence those costs and benefits, we implemented our algorithms and performed several empirical studies. This paper presents the results of those studies. In addition to providing data relevant to the use of our technique, the work provides insights into the use of regression test selection and safe regression test selection techniques generally; in these respects, we believe that the results will be of interest to practitioners. The paper also provides insights into methods for empirically evaluating regression test selection techniques, and highlights questions that should be addressed by subsequent empirical work; in this respect, we believe that the results will be of interest to other researchers.

2 REGRESSION TEST SELECTION

This section discusses selective retest techniques, describes the regression test selection algorithm on which our studies focus, and discusses the cost model that we use to evaluate results.

2.1 Selective Retest Techniques and the Regression Test Selection Problem

Let P be a procedure or program, let P' be a modified version of P, and let T be a set of tests (a *test suite*) created to test P. A typical selective retest technique proceeds as follows:

1) Select $T' \subseteq T$, a set of tests to execute on P'.
2) Test P' with T', establishing the correctness of P' with respect to T'.
3) If necessary, create T'', a set of new functional or structural tests for P'.

- *G. Rothermel is with the Department of Computer Science, Oregon State University, Corvallis, OR 97331. E-mail: grother@cs.orst.edu.*
- *M.J. Harrold is with the Department of Computer and Information Science, Ohio State University, Columbus, OH 43210. E-mail: harrold@cis.ohio-state.edu.*

4) Test P' with T'', establishing the correctness of P' with respect to T''.

5) Create T''', a new test suite and test history for P', from T, T', and T''.

In performing these steps a selective retest technique addresses several 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' that require additional testing. Steps 2 and 4 address the *test suite execution problem*: the problem of efficiently executing tests 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.

2.2 Regression Test Selection Techniques

In earlier work [33], we developed a framework for analyzing regression test selection techniques that consists of four categories: inclusiveness, precision, efficiency, and generality. *Inclusiveness* measures the extent to which a technique selects tests from T that reveal faults in P': A 100 percent inclusive technique is safe. *Precision* measures the extent to which a technique omits tests in T that cannot reveal faults in P'. *Efficiency* measures the space and time requirements of a technique. *Generality* measures the ability of a technique to function in a practical and sufficiently wide range of situations.

We used this framework to compare and evaluate existing code-based regression test selection techniques [33]. This analysis suggested a need for a regression test selection technique that possesses several qualifications. First, the technique must be safe: It must not exclude tests (from the original test suite) that if executed would reveal a fault in the modified program. Second, the technique must be sufficiently precise: It must omit enough unnecessary tests to offset its own expense. Third, the technique must be efficient: Its time and space requirements must be reasonable. Finally, the technique must be general: It must be applicable to a wide class of programs and modifications.

We developed a family of regression test selection algorithms that traverse graphs to select tests [31], [32], [34]. Our most basic algorithm builds control flow graphs[1] for procedure P and modified version P', collects test traces that associate tests in T with edges in the graph for P, and performs synchronous depth-first traversals of the two graphs. During these traversals, the algorithm compares program statements associated with nodes that are simultaneously reached in the two graphs. When the algorithm discovers a pair of nodes N and N' in the graphs for P and P', respectively, such that the statements associated with N and N' are not lexicographically identical, the algorithm selects all tests from T that reached N in P. This approach identifies tests that reach code that is new in, or modified for, P', and identifies tests that formerly reached code that

has been deleted from P. This approach can handle whole programs through its application to pairs of procedures in the program and modified version; however, a second algorithm performs test selection for whole programs more efficiently by building interprocedural control flow graphs and traversing them. Enhanced versions of these algorithms add data dependence information[2] to control flow graphs and use it to select tests more precisely.

We further illustrate our basic algorithm by discussing a simple example of its operation; additional details can be found in [34]. Fig. 1 presents procedure `avg`, and a modified version of that procedure, `avg2`, in which statement S7 has erroneously been deleted and statement S5a has been added. The figure also shows the control flow graphs for the two versions of the program (with differences outlined by dotted boxes) and a test suite with test trace information for the original version of the program. When called with `avg`, `avg2`, and this test trace information, our algorithm first constructs the control flow graphs for the two procedures, and then begins a synchronous traversal of these graphs starting with *entry* and *entry'*. The algorithm marks *entry* "visited" and then considers the successor of *entry*, S1. The algorithm finds that S1' is the corresponding successor of *entry'*, and because S1 is not marked "visited" it compares the statements associated with S1 and S1' for lexicographic equivalence. Because these statements are lexicographically identical the algorithm next visits the nodes that are successors of S1 and S1' and continues its traversal from there. The traversal continues in this manner on S2 and S2' and P3 and P3'; in each case the successors of the nodes have lexicographically identical associated statements. Next, the algorithm must consider two successors of P3: P4 and S9. Traversal through S9 leads to traversal of S10 and S10', and then *exit* and *exit'*, with no differences located or tests selected. When the algorithm reaches P4 and P4' it first seeks a true child of P4' to compare with S5; it finds S5a and compares the statements associated with S5 and S5a. These statements are not lexicographically identical so the algorithm locates the test known to reach S5 in `avg` (t2) and adds it to a list of selected tests. The algorithm now seeks a false successor of P4' and finds S6'; the algorithm next visits S6 and S6'. The algorithm finds the statements associated with the successors of these nodes, S7 and S8', not lexicographically identical, and adds the test that formerly reached S7 (t3) to the list of selected tests. At this point further traversal is unneeded; the algorithm returns test suite {t2, t3}.

An analytical evaluation of our test selection algorithms [34] proves that they are safe for controlled regression testing.[3] Several other safe algorithms now exist [3], [8], [21], [38]; however, our basic algorithm is among the two most

---

1. A control flow graph is a directed graph in which nodes represent program statements and edges represent the flow of control between statements.

2. Let G be the control flow graph for procedure P, and let G contain nodes N_i and N_j, such that N_i and N_j correspond to statements S_i and S_j, respectively, in P. S_j is *data dependent* on S_i if and only if S_i defines some variable v, S_j uses v, and there is a path in G from N_i to N_j such that no node on that path corresponds to a statement in P in which v is defined.

3. Controlled regression testing is the practice of testing a modified version under conditions equivalent to those that were used to test the base program. Controlled regression testing is discussed in detail in [33].

```
Procedure avg

S1.   count = 0
S2.   fread(fileptr,n)
P3.   while (not EOF) do
P4.     if (n<0)
S5.       return(error)
        else
S6.       numarray[count] = n
S7.       count++
        endif
S8.     fread(fileptr,n)
      endwhile
S9.   avg = calcavg(numarray,count)
S10.  return(avg)
```

```
Procedure avg2

S1'.  count = 0
S2'.  fread(fileptr,n)
P3'.  while (not EOF) do
P4'.    if (n<0)
S5a.      print("bad input")
S5'.      return(error)
        else
S6'.      numarray[count] = n
          endif
S8'.    fread(fileptr,n)
      endwhile
S9'.  avg = calcavg(numarray,count)
S10'. return(avg)
```
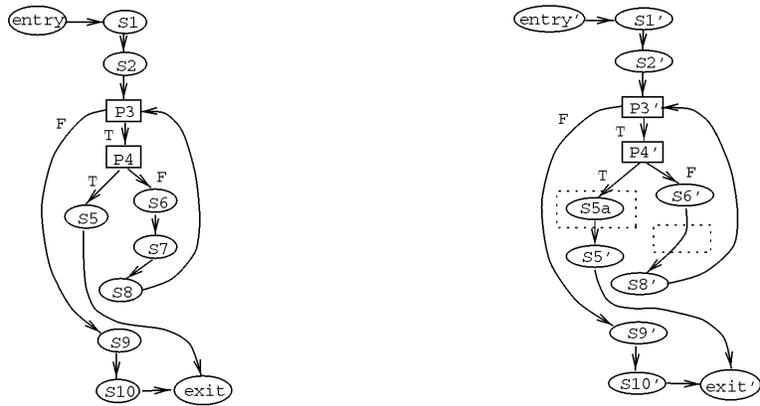
TEST TRACE INFORMATION

test number	input	output	edges traversed
t1	empty file	0	(entry,S1),(S1,S2),(S2,P3),(P3,S9),(S9,S10),(S10,exit)
t2	-1	error	(entry,S1),(S1,S2),(S2,P3),(P3,P4),(P4,S5),(S5,exit)
t3	1 2 3	2	(entry,S1),(S1,S2),(S2,P3),(P3,P4),(P4,S6),(S6,S7) (S7,S8),(S8,P3),(P3,S9),(S9,S10),(S10,exit)

Fig. 1. Procedure `avg`, its modified version, `avg2`, their control flow graphs, and test trace data for `avg`.

precise of these algorithms.[4] Our analysis also shows that our algorithms are at least as general as existing techniques, whether safe or not. Finally, the evaluation shows that in terms of worst-case runtime analysis, our algorithms are comparable to the most efficient other algorithms.

We have implemented several versions of our algorithms; for the experimentation reported in this paper we utilized a hybrid version of our algorithms to produce a tool that we call "DejaVu". DejaVu analyzes whole programs by individually analyzing pairs of procedures from the old and new versions. (We chose this approach to simplify the implementation effort; it produces test selection results equivalent to those of our interprocedural-control-flow-graph based algorithm, although possibly at greater cost [34].) In cases where variable declarations differ, the tool postpones test selection until it reaches occurrences of those variables, and treats the nodes that contain those occurrences as modified. This approach adds precision to the test selection without adding the full cost of, or implementation effort required to support, complete dataflow analysis. To obtain control flow graphs, variable usage data, and test history information, we utilized the Aristotle program analysis system [14]. Given the functionality provided by Aristotle, the DejaVu implementation itself required only 1,220 lines of C code. DejaVu and its interface with Aristotle are described in detail in [31].

2.3 Cost Models for Regression Test Selection Techniques

To evaluate the costs and benefits of regression test selection techniques we require a cost model that accounts for the factors responsible for those costs and benefits. Leung

---

4. Recently, Ball [3] presented a family of control-flow-graph based algorithms that, building on our control-flow-graph based approach, achieve greater precision than our basic algorithm. However, in all empirical studies to date of programs not contrived specifically to demonstrate precision, Ball's algorithms and our basic algorithm have been observed to be equally precise.

and White [26] present such a cost model. Their model considers both test selection and coverage identification, so we adapt it to consider just the cost-effectiveness of a regression test selection technique relative to that of the retest-all approach. For a test selection technique to be more effective than the retest-all approach, the cost of test selection analysis plus the cost of running and validating selected tests must be less than the cost of running and validating all tests. The following inequality summarizes the relationship that must hold:

$$A(T, P, P') + E(T', P') + V(T', P') < E(T, P') + V(T, P').$$

The left side of this inequality represents the costs associated with test selection, where $A(T, P, P')$ is the cost of the analysis required for test selection, $E(T', P')$ is the cost of executing selected tests on the modified program, and $V(T', P')$ is the cost of validating the results of the selected tests. The right side of the inequality represents the costs associated with the retest-all approach, where $E(T, P')$ is the cost of executing all tests on the modified program, and $V(T, P')$ is the cost of validating the results of all tests.

As Leung and White state, this cost model makes simplifying assumptions. The model assumes equivalent test execution costs under regression test selection and retest all, and assumes constant costs for tests. Rosenblum and Weyuker [30] describe additional assumptions: the model assumes that costs can be expressed in equivalent units whereas in practice they can include a mixture of other factors such as CPU time, human effort, and equipment expenditures.

Another factor not accounted for by this regression testing cost model is the cost of leaving undetected faults in modified software. This cost is difficult to quantify in general, but where regression test selection techniques are concerned, a meaningful metric exists: the percentage of tests in T that the test selection technique omits which, if executed on P', would have revealed faults in P'. These *fault-revealing tests* would not be omitted by a retest-all approach, and thus represent an additional cost associated with the use of test selection techniques. One strength of safe regression test selection techniques is that they omit no fault-revealing tests, and thus fare as well in fault detection as the retest-all technique. Questions that must be answered empirically, however, are the extent to which the conditions necessary for safety hold in practice, and the effects on test selection when they do not.

When we apply the foregoing cost model, it is useful to distinguish between two phases of regression testing: a *preliminary phase* and a *critical phase*. The preliminary phase of regression testing begins after a release of a version of the software; during this phase developers enhance and correct the software in preparation for the next release. Meanwhile, testers may plan testing activities, or perform tasks such as test trace collection and coverage analysis that depend solely on the released version of the software. When corrections are complete the *critical phase* of regression testing begins; during this phase regression testing is the dominating activity and typically its time is limited. It is in the critical phase that cost minimization is most important for regression testing. Regression test selection techniques can

exploit these phases by delegating analysis tasks to the preliminary phase; however, it is important to realize that some analysis cannot be performed until after the last modification has been made.

3 EMPIRICAL STUDIES

To empirically investigate the use of safe regression test selection techniques in general and the use of our technique in particular we performed five studies.[5] In this section, we describe each study individually and provide initial discussion of results. In Section 4, after all results have been presented and can be considered simultaneously, we provide further interpretation.

3.1 Study 1

3.1.1 Objectives
The objectives of our initial study were to investigate the feasibility of using our test selection technique and to obtain information that would guide further experimentation.

3.1.2 Subjects
We obtained seven C programs together with a number of modified versions and tests for those programs. These subjects had been used in an earlier study by researchers at Siemens Corporate Research to compare controlflow-based and dataflow-based test adequacy criteria [19].[6] Table 1 lists the subjects. For each subject the table provides the name of the base program, the number of functions in the program, the number of lines of code in the program, the number of modified versions of the program, the number of tests available for the program, and a brief description of the program. As the table shows, the programs range in size from 138 to 516 lines of code, and contain between seven and 21 functions.

The researchers at Siemens constructed tests for these programs by following a process described in [19]; we paraphrase that description here. For each base program they created a large *test pool* containing possible tests for the program. To create these test pools, they first created an initial set of black-box tests "according to good testing practices, based on the tester's understanding of the program's functionality and knowledge of … the code" [19, p. 194], using the *category partition method* and the Siemens Test Specification Language tool [2], [27]. They then augmented this set with manually-created white-box tests to ensure that each executable statement, edge, and definition-use pair in the base program or its control flow graph was exercised by at least 30 tests. For experimentation they selected a number of test suites from each test pool. Their distribution of subjects to us did not include these test suites; for this study we used the entire test pools as test suites.

TABLE 1
SUBJECTS USED IN STUDY 1

Program Name	No. of Functions	Lines of Code	No. of Versions	Test Pool Size	Description of Program
totinfo	7	346	23	1,052	information measure
schedule1	18	299	9	2,650	priority scheduler
schedule2	16	297	10	2,710	priority scheduler
tcas	9	138	41	1,608	altitude separation
printtok1	18	402	7	4,130	lexical analyzer
printtok2	19	483	10	4,115	lexical analyzer
replace	21	516	32	5,542	pattern replacement

The researchers at Siemens sought to study the fault-detecting effectiveness of various coverage criteria. Therefore, they created *faulty* modified versions of the seven base programs by manually seeding those programs with faults, usually by modifying a single line of code in the base version. In a few cases, they modified between two and five lines of code. Their goal was to introduce faults that were as realistic as possible, based on their experience with real programs. To obtain meaningful results, the researchers retained only faults that were "neither too easy nor too difficult to detect" [19, p. 196], which they defined as being detectable by at least three and at most 350 tests in the test pool associated with each program. Ten people performed the fault seeding, "mostly without knowledge of each other's work" [19, p. 196].

For regression testing experiments, we can view the base and faulty modified versions of the subjects in either of two ways. First, we can consider the faulty modified versions of base programs to be ill-fated attempts to create modified versions of the base programs. We can then study the effectiveness of `DejaVu`, and the fault-detection effects of using `DejaVu`, on those faulty modified versions. Alternatively, we can consider each base program to be a corrected version of a family of faulty earlier versions, and study the effort required to regression test the corrected base program. It is an interesting characteristic of our test selection algorithms (though not of all test selection algorithms) that they select the same tests under either of these interpretations. To study the fault-detection abilities of test selection techniques, however, the first interpretation is necessary; thus, for our studies, we utilized that interpretation.

The programs we used in this study are not large, and the modifications involve only faults that yield relatively low fault-detection rates. Furthermore, our use of the entire test pools as test suites does not reflect realistic testing practices. These are primarily *external threats to validity*, that limit our ability to generalize our results to industrial practice. Similarly, the use of entire test pools as test suites may provide savings in test execution time that cannot be achieved on more practical test suites. To examine the possible effects of these threats, our subsequent studies vary the characteristics of subjects, modifications, and test suites; we then examine the consequent variations in results.

As subjects for an initial study, however, these programs, versions, and test suites had several advantages. First, we could easily obtain them, and we were able to use our prototype tools on them after only minor modifications; thus, the subjects let us quickly address the study's objectives. Second, the seeded faults do model real faults and, as such,

yield a set of faulty versions that could occur in practice. Moreover, these faulty versions were created by persons other than us, reducing the potential for bias. Finally, the subjects are suitable for use in controlled studies.

3.1.3 Procedure

We used the following procedure for this study. Given program P, versions $P_1 \ldots P_n$, and test pool T, we

1) used `Aristotle` to construct the control flow graph for P,
2) used `Aristotle` to instrument P,
3) ran all tests in T on P, collecting trace information, and capturing outputs for use in validation,
4) built test history H from the trace information,
5) for each version P_i, we
   a) used `Aristotle` to build the control flow graphs for P_i,
   b) ran `DejaVu` on P, P_i, and H, relevant to P_i,
   c) ran and validated outputs for all tests in T on P_i,
   d) ran and validated outputs for all selected tests on P_i,
   e) ran and validated outputs for all nonselected tests on P_i.

We used the Unix `time` command to measure the work required to perform each of these steps, and recorded the "real" (wall clock) time reported by that command.[7] To measure the cost of the retest-all approach we used the time corresponding to Step 5c. To measure the cost of performing regression test selection using `DejaVu`, we added the times corresponding to Steps 1, 5a, 5b, and 5d. Although we did not do so here, we could trade time for space by performing Step 1 in the preliminary phase, saving the resultant control flow graphs, and using the precomputed control flow graphs in Step 5b, avoiding their calculation during the critical phase.

To validate outputs in Steps 5c, 5d, and 5e, we used the base version of the program as an oracle: for a given modified version and test we compared the output of the base version on that test to the output of the modified version on that test, and classified the test as fault-revealing if those outputs differed.

Steps 2, 3, and 4 of this procedure involve work that need not be performed during the critical period. We nei-

---

7. CPU time could also serve as an indicator of cost; however, we believe that the wall-clock time spent on testing, by humans or machines, is a more appropriate metric.

ther count the time spent performing these steps nor would it make sense to do so—to run all tests on an instrumented program during the critical phase in order to determine which tests we need to run would not be cost-effective. Instead, Steps 2, 3, and 4 are initially performed (and ideally, automated) during testing of the base version (which must be tested with retest-all because it is the first version). Then, on subsequent versions, incremental updates required to keep the information current can be performed during the preliminary period.[8]

Step 5e of this procedure lets us determine whether De-jaVu omitted any fault-revealing tests that a retest-all technique would not omit.

We performed this procedure on each of the seven base programs and collected data on each run. We performed all runs on a Sun Microsystems SPARCstation 1+ with 24 MB of virtual memory.[9] Our testing processes were the only active user processes on the machine.

3.1.4 Results

Fig. 2 displays test selection statistics for Study 1. For each of the seven base programs, the figure contains a graph that shows the percentage of tests selected by DejaVu for each version of that program. In each graph, the horizontal axis shows the versions of the program and the vertical axis shows the percentage of tests selected by DejaVu. For the seven programs, on average over the modified versions of the programs, DejaVu selected the following percentages of tests: 72.6 percent for totinfo, 65.3 percent for schedule1, 93.4 percent for schedule2, 67.4 percent for tcas, 55.8 percent for printtok1, 33.7 percent for printtok2, and 43.3 percent for replace. On average overall, DejaVu selected 54.3 percent of the tests in the test pools.

Fig. 2 also shows that test selection results for DejaVu varied widely between individual programs and versions. For example, replace has a test pool of 5,542 tests, of which DejaVu selected, on average, 2,398 (43.3 percent); however, the selected test suites vary in size from 52 to 5,520, with no range of sizes predominant. In contrast, schedule2 has a test pool of 2,710 tests, of which DejaVu selected, on average, 2,660 (92.4 percent). On eight of the 10 modified versions of this program, DejaVu selected over 95 percent of the tests.

The fact that a test selection technique reduces the number of tests that must be run at regression testing time does not guarantee that the technique will be cost-effective. As the regression testing cost model presented in Section 2 indicates, regression test selection techniques can produce savings only when analysis costs plus the cost of running and validating selected tests is less than the cost of running and validating all tests. To measure the costs of analysis, test execution, and test validation in this study, we rely on time statistics. For each of the seven subject programs, Table 2 shows the times in seconds, averaged over all modified
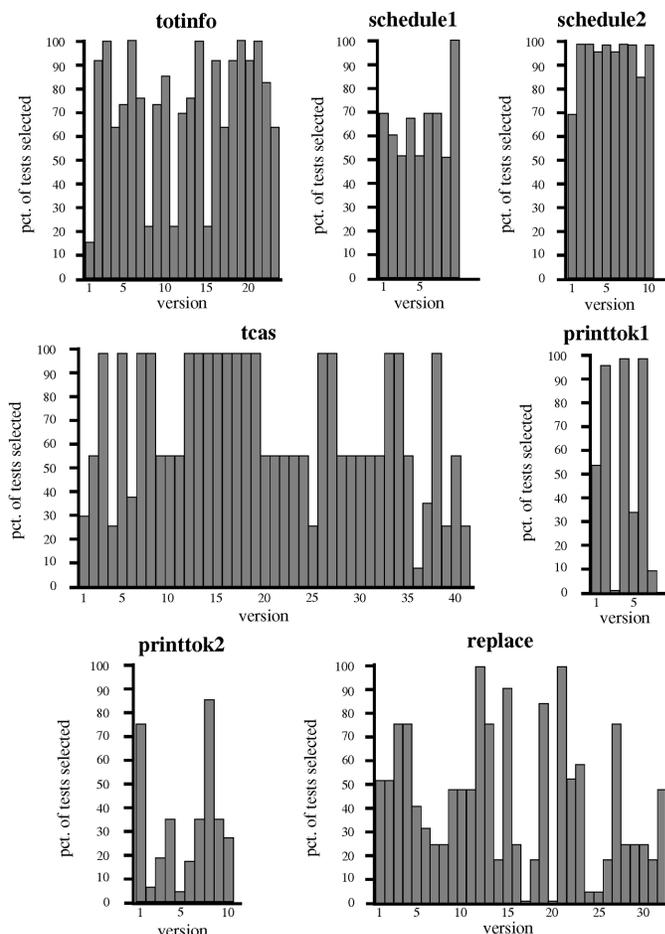

Fig. 2. Test selection statistics for Study 1.

versions of the program, required to accomplish various tasks. The columns show (from left to right) the time required to run and validate output for all tests on the modified versions of the programs, the time required to build control flow graphs for the base and modified versions and run DejaVu on those control flow graphs, the time required to run and validate output for just the selected tests, the time saved by test selection, the percentage of total time saved by DejaVu, and the *break-even value*, which we define momentarily. The bottom row of the table lists averages over all 132 versions.

In this study, on average, DejaVu reduced testing effort (time) by 42 percent, from 14 min, 27 sec to 8 min, 21 sec. Like test selection results, however, time savings varied widely over programs and modified versions. To illustrate the latter, Table 3 shows time statistics for the individual versions of printtok1. Over the seven versions, the average savings in time was 41 percent. On four modified versions, the savings exceeded this average, but on three modified versions there were no savings—in fact, on these three versions test selection actually increased total regression testing time by 13, 14, and 16 sec, respectively. Similar results occurred on other programs and versions. This wide variation raises the question of whether it is possible to predict, in advance of running a test selection tool, whether or not that tool is likely to produce savings. We discuss this issue further in Section 4.

---

8. An alternative approach applies information gathered on the base version to a succession of modified versions. Although not safe, this approach may support useful test selection in cases where incremental update of test information is not cost-effective. Further study of the cost-benefits tradeoffs of this approach is necessary.

9. SPARCstation is a trademark of Sun Microsystems, Inc.

TABLE 2
STUDY 1: EXECUTION TIMES AND SAVINGS (SECONDS) FOR EACH PROGRAM AVERAGED OVER ITS SET OF VERSIONS

Subject	Time to Run All Tests	Time to Perform Analysis	Time to Run Tests Selected	Time Saved	Pct. of Time Saved	Break-Even Value
totinfo	339	22	246	71	21	68
schedule1	815	26	536	253	31	84
schedule2	843	28	789	26	3	90
tcas	429	15	288	126	29	56
printtok1	1,253	44	698	511	41	145
printtok2	1,232	47	413	772	63	155
replace	1,633	59	704	870	53	200
all versions	867	33	468	366	42	110

The bottom row lists averages over all 132 versions.

TABLE 3
STUDY 1: EXECUTION TIMES AND SAVINGS (SECONDS) FOR VERSIONS OF **PRINTTOK1**

Version	Time to Run All Tests	Time to Perform Analysis	Time to Run Tests Selected	Time Saved	Pct. of Time Saved	Break-Even Value
1	1,294	43	664	547	4	137
2	12,562	44	1,222	−13	0	144
3	1,252	44	12	1,198	95	145
4	1,241	44	1,223	−14	0	146
5	1,238	44	426	783	62	147
6	1,242	44	1,226	−16	0	146
7	1,245	44	114	1,096	87	146
Average	1,253	44	698	511	41	145

In the tables, the *break-even value* estimates the number of tests that DejaVu must eliminate in order to offset the analysis costs and provide cost (time) savings in regression testing. The break-even value is calculated by taking the ratio of analysis time to the time required to execute and validate a single test. More formally, given program P, version P', and test suite T containing $|T|$ tests, such that the time required to run all tests in T on P' is $E(T, P') + V(T, P')$ and the time required to perform the analysis for test selection on P, P', and the test history for T is $A(T, P, P')$, the break-even value B for P, P', and T is defined as:

$$B(P, P', T) = \frac{A(T, P, P') * |T|}{E(T, P') + V(T, P')}$$

(This number is an estimate because it assumes that the cost of executing a test is uniform across all tests.) In this study, for our subject programs, the break-even values range from an average of 56 for tcas, to 200 for replace, for an overall average (over all 132 versions of the programs) of 110.

Our measurements of the fault-detection ability of DejaVu showed that in this study, DejaVu never omitted any fault-revealing tests. In every case, DejaVu's ability to select tests that detect faults in the modified program was equivalent to that of the retest-all technique.

The average, overall time savings achieved by DejaVu in these studies amounted to a little over six minutes. By absolute measures such savings are insignificant; however, regression testing of large-scale, commercial software systems can require hours, days, or weeks of effort, and may involve considerable human labor. In such cases, even a small reduction in testing time could be worthwhile, and a 42 percent reduction in testing time could be significant. Such savings depend, however, on whether DejaVu can

produce analogous results on larger-scale subjects. Our fourth and fifth studies investigate this question.

3.2 Study 2

3.2.1 Objectives

In practice, we do not expect test suites for programs such as those we used in Study 1 to contain thousands of tests; thus, it is reasonable to ask how test selection results and cost-effectiveness would be affected by the use of more typical test suites.

The objectives of our second study, therefore, were to investigate whether the results of Study 1 extend to the case where realistic code-coverage-based test suites, rather than test pools, are utilized.

3.2.2 Subjects

For this study, we used the same base programs and modified versions that we used in Study 1. To obtain test suites for these programs, we used the test pools for the base programs and test coverage information generated in the first study to generate 1,000 branch-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 used the C pseudo-random-number generator "rand," seeded initially with the output of the C "time" system call, to obtain integers that we treated as indexes into T_p (modulo $|T_p|$). We used these indexes to select tests from T_p; we added each test t to T only if t added to the cumulative branch coverage of P achieved by the tests added to T thus far. We continued to add tests to T until T contained at least one test that would exercise each executable branch in the base program. Table 4 lists the average sizes of the branch-coverage-adequate test suites generated by this procedure for the subject programs.

TABLE 4
BRANCH-COVERAGE-ADEQUATE TEST SUITES
GENERATED FOR STUDY 2

Program Name	Test Pool Size	Test Suite Average Size
totinfo	1,052	7
schedule1	2,650	8
schedule2	2,710	8
tcas	1,608	6
printtok1	4,130	16
printtok2	4,115	12
replace	5,542	19

These test suites are not minimal: for each test suite there may be a proper subset of that suite that is also branch-coverage-adequate. Thus, the results we obtain with these test suites may not generalize to minimal coverage suites. Nevertheless, we believe that our method of constructing these test suites—successively adding tests to each suite until coverage is achieved—represents a realistic process for generating coverage-adequate test suites.

3.2.3 Procedure

We used the following procedure for this study. Given program P, versions $P_1 \ldots P_k$, and universe T_U of test suites, we

1) used **Aristotle** to construct the control flow graphs for P,
2) used **Aristotle** to instrument P,
3) for each test suite $T \in T_U$, we
   a) ran all tests in T on P, collecting trace information, and capturing outputs for use in validation,
   b) built test history H from this trace information,
   c) for each modified version P_i of P, we
      i   used **Aristotle** to build the control flow graphs for P_i,
      ii ran **DejaVu** on P, P_i, and H,
      iii ran and validated outputs for all tests in T on P_i,
      iv ran and validated outputs for all selected tests on P_i,
      v   ran and validated outputs for all nonselected tests on P_i.

As in Study 1, to validate output for a given modified version and test, we used the base version as an oracle: we compared the output of the base version on that test to the output of the modified version on that test. As in Study 1, we used the Unix **time** command to measure the cost of performing the associated work, recording the "real" (wall clock) time reported by that command. To measure the cost of retest-all we used the time corresponding to 3ciii; to measure the cost of **DejaVu** we added the times corresponding to 1, 3ci, 3cii, and 3civ. We performed this procedure on each of the seven base programs and collected data on each run. We performed all runs on the same Sun Microsystems SPARCstation 1+ used in Study 1. Our testing processes were the only active user processes on the machine.

3.2.4 Results

Fig. 3 displays test selection statistics for Study 2. The figure contains a separate graph for each of the seven subject pro-

grams. Each graph plots percentages of tests selected (vertical axis) against modified versions of the base program (horizontal axis). Results for each version are depicted by a *box plot*—a standard statistical device for representing data sets [20]. In each box plot, the dashed crossbar represents the median percentage at which tests were selected over the 1,000 test suites of that version. The box shows the range of percentages in which the middle 50 percent of the test selection results occurred (the *interquartile range*). Often, this range is evenly partitioned by the crossbar, indicating an even distribution of the data in the interquartile range about the median. In other cases, the data is *skewed* to one side of the crossbar, indicating an uneven distribution. In yet other cases, only the crossbar appears, indicating that the box has length 0: At least half of the results were equivalent to the median result. The *whiskers* that extend below and above the box indicate the range over which the lower 25 percent and upper 25 percent of the data, respectively, occurred. Consider, for example, the graph for **totinfo**, displayed in the upper left corner of the figure. The box plot in the leftmost column of this graph depicts test selection results for version 1 of **totinfo**, for the 1000 test suites of that program. The box plot shows that the median test selection for that version occurred at 22 percent, with half of the test selection results between 14 and 33 percent and with the interquartile data skewed slightly toward 100 percent; the plot also shows that the test selection ranged between 9 and 80 percent overall, with the complete data skewed significantly toward 100 percent.

The graphs illustrate several facts about test selection. As in Study 1, test selection results vary widely between individual programs and versions. Some programs, such as **replace**, exhibit a wide range of test selection results between versions. Other programs, such as **schedule2**, exhibit similar results for most versions. Within individual versions, test selection results also vary between test suites: on average over all 132 versions of the seven programs, the interquartile range covers a spread of 9.4 percent. As the median test selection percentage approaches one of the extremes (i.e., 0 or 100 percent), the interquartile range frequently becomes smaller, which indicates less variability in the results, and becomes more skewed in the direction opposite to the extreme, which indicates a tendency to vary more away from the extreme than toward it.

Overall, these test selection results are comparable to those observed in Study 1, indicating that the selectivity results achieved on the test pools used in that study do generalize to the coverage-based test suites that we generated for this study.

Table 5 shows the average time statistics that we collected for the seven programs in this study. In every case, the time required to execute and validate all tests is small, and the time required to analyze the base program and modified version exceeds the time required to run and validate all tests. In this case, the break-even values (the number of tests that must be omitted in order to realize savings) are lower than those calculated for Study 1, because of the lower costs of analysis associated with the reduced-size test suites. Despite this fact, for each program the break-even value exceeds the number of tests in the test
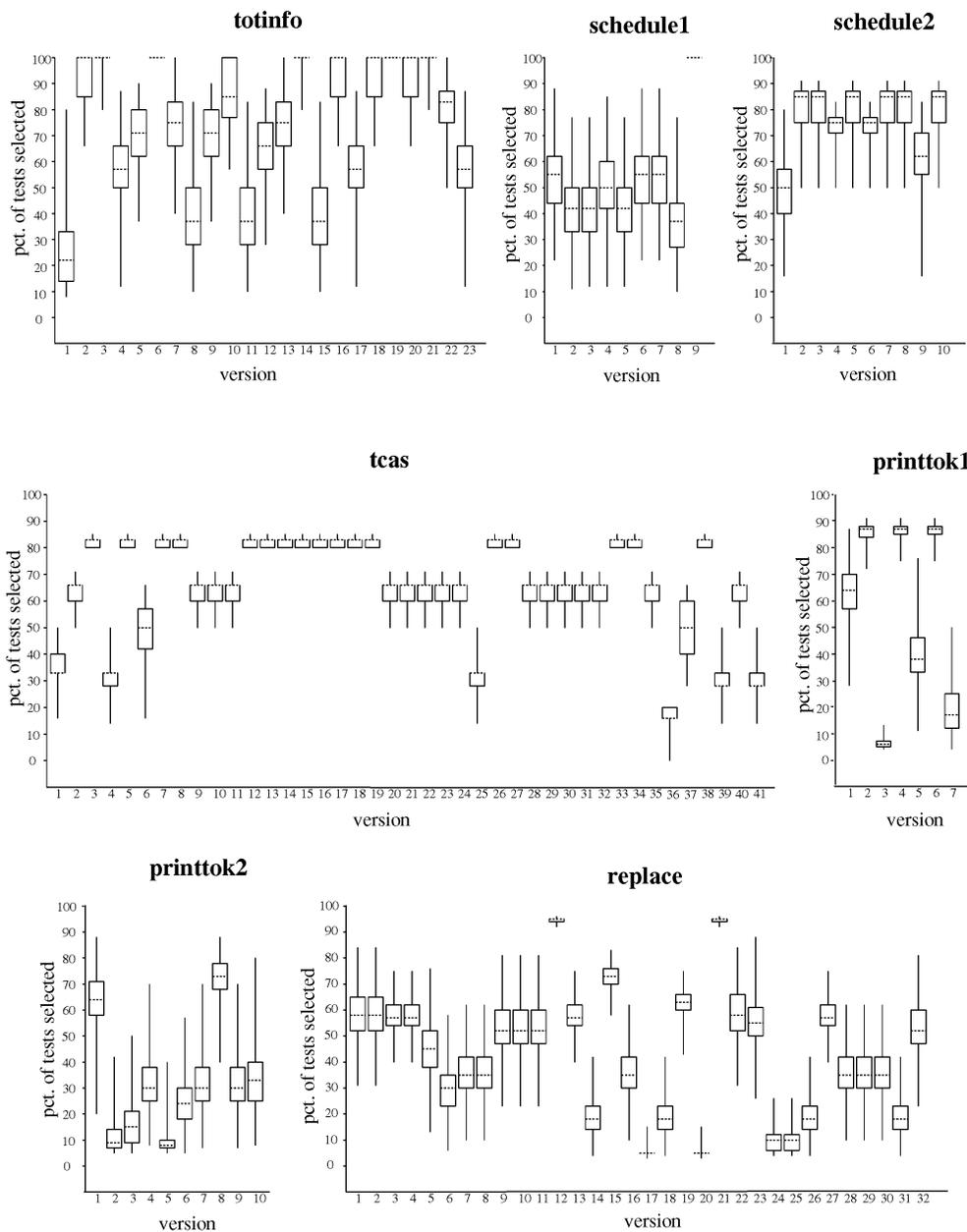
Fig. 3. Test selection statistics for Study 2.

TABLE 5
STUDY 2: EXECUTION TIMES AND SAVINGS (SECONDS) FOR EACH PROGRAM
AVERAGED OVER ITS SET OF VERSIONS

Subject	Time to Run All Tests	Time to Perform Analysis	Time to Run Tests Selected	Time Saved	Pct. of Time Saved	Break-Even Value
totinfo	2	17	2	−17	0	53
schedule1	2	18	1	−19	0	58
schedule2	2	16	2	−16	0	51
tcas	1	11	1	−11	0	41
printtok1	3	21	1	−19	0	69
printtok2	2	16	1	−15	0	53
replace	5	26	2	−23	0	88
all versions	2	17	1	−16	0	59

The bottom row lists averages over all 132 versions.

suites for the program. In cases such as these, test selection techniques—no matter how successful at reducing the number of tests that must be run—cannot provide savings.

In this analysis it is important to remember that break-even values depend on the cost of executing individual tests. For the subjects examined in this study, tests are inexpensive, requiring only between 0.27 and 0.32 sec apiece to execute and validate (on average). If these tests required 10 times this effort (that is, if they required a mere 3 to 4 sec apiece), the break-even values for these subjects would be reduced by a factor of 10, and test selection would frequently have been cost-effective. Such an increase in effort could occur if, for example, some portion of the test execution and validation effort were not automated.

Finally, as in Study 1, our test selection technique did not omit any tests from the original test suites that, if executed, exposed faults in the modified version. In every case, `DejaVu` selected tests that detected faults as well as the retest-all technique.[10]

3.3 Study 3

3.3.1 Objectives

In practice, test suites may not be designed to meet code-based adequacy criteria. In this case, the code coverage achieved by test suites may be distributed among program components differently than the coverage achieved by code-coverage-adequate suites, and this may affect test selection results.

The objective of our third study, therefore, was to investigate the effects on test selection of using non-coverage-based test suites instead of coverage-based test suites.

3.3.2 Subjects

To address the foregoing objective in a controlled fashion, we required subjects equivalent to the subjects used in our second study in all respects except for the test suites. Thus, we used the same programs and versions as those used in the second study, and we generated test suites of the same size as those used in that study, except that we did not generate those suites for coverage.

We chose random test selection from test pools as our mechanism for creating non-coverage-based test suites. To obtain the required test suites, for each branch-coverage-adequate test suite T^C used in Study 2, we created a test suite T^R containing the same number of tests as T^C, made up of tests randomly selected from the test pool without regard for coverage. More precisely, we used the following procedure, which we repeated for each of the seven subject programs. Given program P, $S^C = T_1^C, T_2^C, \ldots T_{1,000}^C$ the set of 1,000 coverage-adequate test suites for P created for Study 2, and U the test pool for P, we

1) let S^R be the set of 1,000 non-coverage-based test suites to be created, initially empty
2) for $k = 1$ to 1,000

    a)   let T_k^R be a test suite to be created, initially empty

    b)   while $\left| T_k^C \right| \neq \left| T_k^R \right|$

        i   select a test t randomly from U

        ii $T_k^R = T_k^R \cup t$

    c) $S^R = S^R \cup T_k^R$

Test suites created by this procedure may not be representative of non-coverage-based test suites created in practice, which more typically are created by some nonrandom process such as that of covering functional requirements. Thus, test selection results obtained on these test suites may not reflect results that would be obtained on realistic test suites. However, these test suites let us investigate, in a controlled manner, the effects on test selection of using non-coverage-adequate test suites instead of coverage-based test suites. In our fourth and fifth studies, we utilize realistic non-coverage-based test suites.

3.3.3 Procedure

This study utilized the same procedure as Study 2. We applied the procedure to each of the seven subject programs with its modified versions, and collected data on each run, on the same machine and under the same operating conditions as in Study 2.

3.3.4 Results

Fig. 4 displays test selection results for Study 3, using the same format we used to depict the results of Study 2. Again, results vary widely between programs, modified versions, and test suites. Again, as the median test selection approaches an extreme, the interquartile range tends to become smaller.

The data also reveals additional trends in comparison to the results of Study 2, where individual test suites are concerned. First, consider cases in which the median test selection for the coverage-based test suites is between 20 and 80 percent. In these cases, most of the boxplots for non-coverage suites exhibit greater spread than the corresponding boxplots for coverage-based test suites—an effect reflected in an increased size of interquartile ranges and an increased length of whiskers. This result is particularly apparent for `schedule1`, where it holds for all versions other than version 9.

Second, in cases in which the median test selection result for the coverage-based test suites is above 80 percent or below 20 percent, the interquartile range for non-coverage suites is typically smaller, and the data is typically skewed significantly more toward 50 percent, than for the corresponding coverage suites. This result is particularly apparent for `schedule2`, where, for six of the 10 versions of the program, median test selection for the coverage-based test suites exceeds 80 percent. For all six of these versions, the interquartile range for non-coverage based suites is significantly smaller (all data points in the range occur at 100 percent) than the interquartile range for the corresponding coverage-based suites.

Table 6 provides additional data, showing the average interquartile ranges for coverage-based suites and non-coverage suites on each of the seven subject programs. The data shows that overall, the average interquartile range for noncoverage suites exceeds the average interquartile range

---

10. This result does not imply that either `DejaVu` or retest-all always revealed the fault in each modified version, because not every coverage-based test suite revealed each fault.
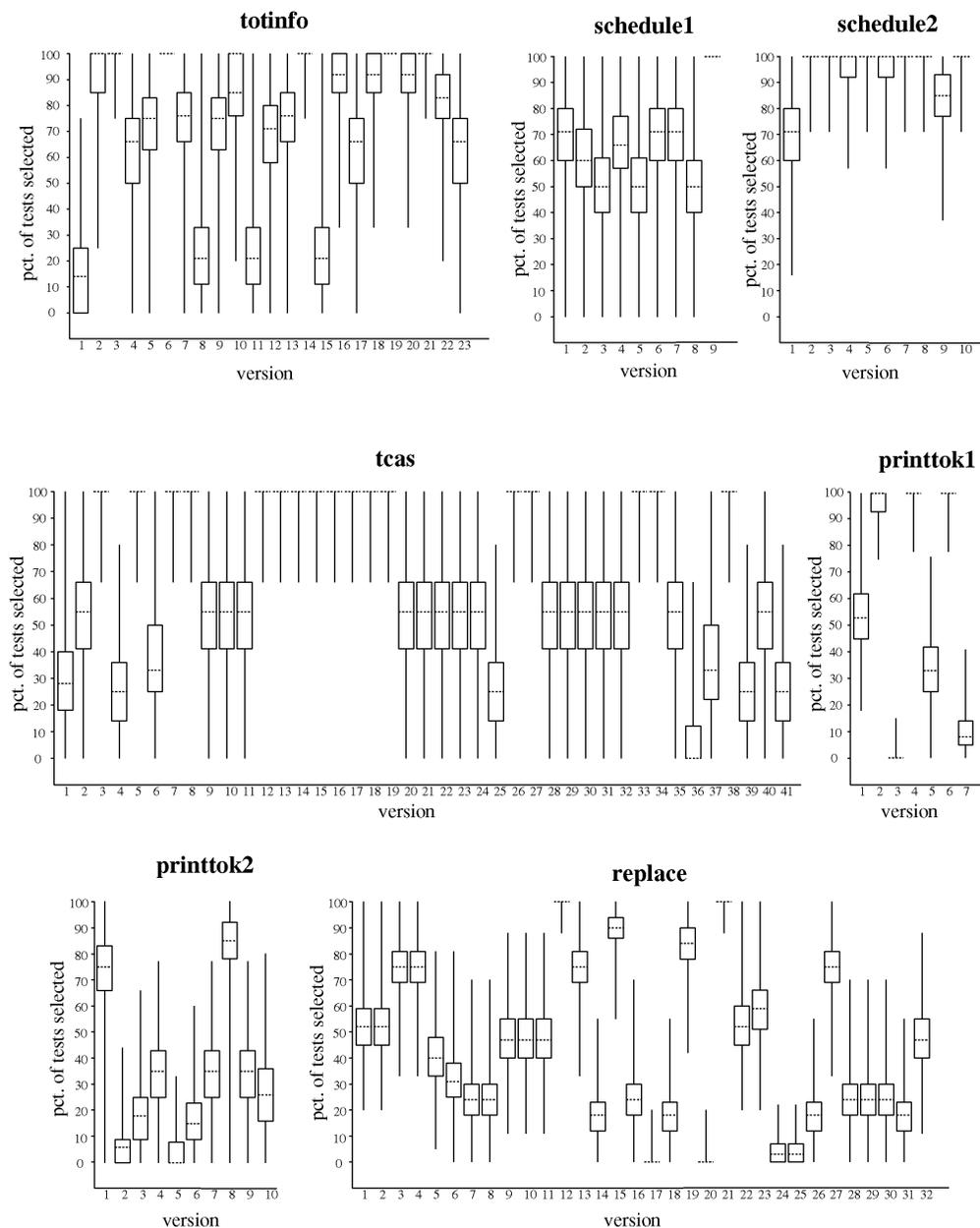
Fig. 4. Test selection statistics for Study 3.

TABLE 6
AVERAGE INTERQUARTILE RANGES FOR COVERAGE-BASED
SUITES AND NON-COVERAGE-BASED SUITES
FOR THE SUBJECT PROGRAMS

Program Name	Average Interquartile Range (coverage-based suites)	Average Interquartile Range (non-coverage-based suites)
totinfo	5.5	14.0
schedule1	15.4	18.5
schedule2	12.1	5.1
tcas	13.5	15.8
printtok1	9.4	7.3
printtok2	11.2	15.3
replace	7.3	11.6
all versions	9.5	13.5

for non-coverage-based suites by 43 percent. Not all of the seven programs contribute to this effect; schedule2 in particular differs, exhibiting a substantially smaller interquartile range for non-coverage-based suites than for coverage-based suites. This difference reflects the preponderance of test selection results in excess of 80 percent for that program.

Given our procedure for obtaining subjects for and performing this study, we know that these differences in the spread and skew of test selection data are caused by differences between the coverage-based and non-coverage-based test suites. The following discussion attempts to further explain the causes of those differences. First, in cases in which DejaVu selects a small percentage of the coverage-based tests, it is because the code changes in the modified version involve code that is relatively infrequently executed. In such cases, the modified code may be even less frequently executed, on average, by random program inputs,

resulting in lower selection statistics for the non-coverage-based suites. Second, in cases in which `DejaVu` selects a large percentage of the coverage-based tests, it is because the code changes in the modified version involve code that is relatively frequently executed. In such cases, the modified code may be even more frequently executed by random program inputs, resulting in higher selection statistics for the non-coverage-based suites. Finally, in cases in which `DejaVu` selects a midrange percentage of the coverage-based tests, randomly chosen inputs may exhibit greater variance in code coverage than inputs utilized in coverage-based test suites, in some cases duplicating coverage, and in others omitting coverage, resulting in greater variance in test selection statistics for the non-coverage-based suites. Coverage suites, conversely, reduce variance in coverage by mandating inclusion of tests through hard-to-reach code, and excluding tests that only duplicate coverage. As we shall discuss in Section 4, these results have implications for test-suite-design for regression testability.

Other results obtained in this study are similar to those obtained in Study 2. Time statistics on test-execution and test-selection costs, shown in Table 7, are nearly identical to those displayed in Table 5. This is not surprising given the equivalence of subjects and test suite sizes across the two studies. Again, for these subjects, with their test execution and analysis times and consequent break-even values, test selection is not cost-effective. However, just as in Study 2, if test execution and validation were more expensive, the reductions in test suite size produced by `DejaVu` could frequently have resulted in savings.

Finally, in this study, our test selection technique again did not omit any tests from the original test suites that, if executed, exposed faults in the modified version.

3.4 Study 4

3.4.1 Objectives

The objective of our fourth study was to investigate the results of applying our test selection technique to a nontrivial software system.

3.4.2 Subject

Table 8 describes the subject we used in this study. The base program, `player`, is the largest subsystem in the software distribution for the Internet game `Empire`. As the table indicates, the base version contains 766 functions (all written in C) and 49,316 lines of code. The `player` program 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. The `player` program is invoked and left running on some system; a user then communicates with `player` by running a small `client` program that receives the user's inputs and passes them as commands to `player`. When a command is received by `player`, code in the event loop invokes a routine that processes the command—possibly invoking many more routines to do so—and then waits to receive the next command. As it processes 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; most changes involve the `player` subsystem. We located a "base" version of `player` for which five distinct modified versions, which were created independently by various coders for various purposes, were available; that base version and those modified versions constitute the source and versions we used in this study. Table 9 describes the five versions of `player` that we located. As the table illustrates, the versions vary in terms of lines-of-code and functions modified. Note that these versions do not form a sequence of modifications of the base program; rather, each is a unique modified version of the base version.

TABLE 7
STUDY 3: EXECUTION TIMES AND SAVINGS (SECONDS) FOR EACH PROGRAM AVERAGED OVER ITS SET OF VERSIONS

Subject	Time to Run All Tests	Time to Perform Analysis	Time to Run Tests Selected	Time Saved	Pct. of Time Saved	Break-Even Value
totinfo	3	19	2	−18	0	59
schedule1	3	18	2	−17	0	58
schedule2	3	17	3	−17	0	55
tcas	2	14	1	−13	0	52
printtok1	5	20	2	−17	0	66
printtok2	4	19	1	−16	0	62
replace	6	22	3	−19	0	75
all versions	4	18	2	−17	0	58

The bottom row lists averages over all 132 versions.

TABLE 8
SUBJECT USED IN STUDY 4

Program Name	No. of Functions	Lines of Code	No. of Versions	Test Pool Size	Description of Program
player	766	49,316	5	1,033	transaction manager

TABLE 9
MODIFIED VERSIONS USED IN STUDY 4

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

The `player` program 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, the `player` program 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 tests available for `player`. To construct a realistic test suite, we used the `Empire` information files, which describe the 154 commands that are recognized by `player` and describe the parameters and special effects associated with each command. We treated these information files as an informal specification for the system and used them to construct a suite of tests for `player` that exercises each parameter, special effect, and erroneous condition described in the files. Because the complexity of commands, parameters, and effects varies widely across the various `player` commands, this process yielded between one and 30 tests for each command, and ultimately produced a test pool of 1,033 tests. To avoid a possible source of bias, we constructed this test suite prior to examining the code of the modified versions.

Each test that we created by the foregoing process consists of a sequence of between one and 28 lines of ASCII text, and constitutes a sequence of inputs to the `client` program; the `client` feeds these inputs to `player`. For each sequence of inputs to be valid and test the `player` functionality that it is targeted to test, the game database must be initialized to a specific state prior to applying that sequence of inputs. Thus, to automate the regression testing process, we created a script that iterates through the tests, and for each test performs the following steps:

1) restore the required start state of the database,
2) invoke the `player` program as a background process (this process remains running in the background until explicitly killed),
3) invoke the `client` program and issue the sequence of inputs that constitutes the test (note that this sequence always ends in the issuance of a "quit" command that causes the client to terminate), saving for use in validation any outputs that are returned,
4) kill the `player` program process,
5) save the contents of the database for use in validation,
6) compare output and database contents with those archived for the base version.

Although the particulars of this testing process apply only to the `player` program, analogous processes are used to automate the regression testing of a variety of other software systems: the system is initialized, test inputs are applied, and outputs are captured and compared to previously captured outputs. More generally, in this and other testing processes each test requires effort to execute and validate. That effort may take the form outlined above or some other form; the important factors to consider where regression test selection is concerned are not the particular processes by which tests are executed and validated, but rather the costs of analysis, test execution, and validation, the ability of the test selection technique to reduce the number of tests that must be run, and the ability of the regression testing activity to reveal faults. Thus, our use of this particular testing process does not represent a threat to validity.

The test suite generated for `player` contains black-box, functional tests, rather than tests designed for code coverage. In view of our discoveries in the preceding two studies, we cannot claim that results obtained using this test suite will generalize to code-coverage-based suites. Our test suite may, in fact, be redundant in terms of code coverage, and minimizing the test suite for code coverage might reduce its size. However, the test suite *is* nonredundant in the sense that no two tests cover the same functional requirements, and in practice, few testers would minimize such a test suite. Thus, we believe that, as created, this test suite is representative of a large class of test suites utilized in practice, and that the results we obtain using this test suite are indicative of results that could be expected to occur in practice.

3.4.3 Procedure

Because of limitations in our prototype code instrumenter, we could not instrument all of the `player` code; thus, we could not obtain test history information for `player`.[11] Because `DejaVu` requires test history information, we could not use the tool to select tests for `player`. We were able, however, to simulate the test selection effects of `DejaVu`. Because the simulation process may be useful for future experimentation with regression test selection, we describe it in detail, as follows.

For each modified version `player'` of `player`, we used the Unix `diff` utility to locate differences between `player` and `player'`. We then edited `player'` and instrumented it as follows. At each executable code location where the two versions differed, we inserted the function call "`dejavu()`". In cases where variable declarations differed, we found the locations in the code where those variables occurred, and instrumented those locations as if they contained modified code—this approach produces results equivalent to those achieved by the implementation of `DejaVu` that we utilized in Studies 1 through 3. The `dejavu()` function, which we supplied, opens a file `result`, writes the phrase "selected" to that file, and closes it. We also edited function `main` in `player'`, and inserted, as the first executable statements in that function, code that opens the `result` file, writes the phrase "not selected" to that file, and closes it. We then

---

11. These limitations are due only to our particular code instrumenter, which is a prototype and is not sufficiently robust to accommodate the variety of C usages found in `player`.

compiled and linked `player'`. Given these edits, when our instrumented `player'` is invoked, it immediately writes "not selected" to the `result` file, and subsequently, if and only if modified code is encountered, it invokes the `dejavu()` function. In that case, the `dejavu()` function overwrites the results file with the phrase "selected." Because the file is opened and closed anew on each call to the `dejavu()` function, the file always contains exactly one line; after the file contains the phrase "selected" it retains that phrase until the program terminates.

Given the foregoing instrumentation, when we run a test *t* on `player'`, we know that *t* is modification-traversing and would be selected by `DejaVu` if and only if the `results` file contains the phrase "selected" when `player` terminates. The simulation requires us to run all tests in order to determine which tests `DejaVu` would select, and thus is of interest only as a simulation; however, the simulation lets us measure test selection results in the absence of the test history information required by `DejaVu`.

We were able to precisely measure most of the costs associated with test selection for `player`. First, we were able to precisely measure the cost of running the tests for `player` and the cost of running selected tests, by running those tests and timing the runs. Second, we were able to precisely determine the cost of constructing control flow graphs for `player` and its modified versions—this constitutes one component of the cost of the analysis performed by `DejaVu`.[12] In the absence of test history information, however, we could not precisely determine the analysis time (beyond the time required for control flow graph construction) required by `DejaVu`. Instead, we estimated the rest of the tool's cost. To obtain this estimate we created a modified version of `DejaVu`, `DejaVuSim`, that performs all graph traversals and comparison operations performed by `DejaVu` (on precomputed control flow graphs), and on completion of its traversal, performs *n* set union operations on sets containing *k* tests, where *n* and *k* are parameters supplied at invocation. By running `DejaVuSim` on `player` with `player` itself, we forced the tool to completely traverse all control flow graphs for `player`, providing an overestimate of the cost of traversals. By setting *n* to the number of `dejavu()` calls present in the modified version, and *k* to the number of tests known to be selected for that version, we forced the `DejaVuSim` tool to perform the maximum number of set operations that it could have performed in practice.[13]

12. For an initial version of this study, summarized in [34], we could only estimate control flow graph construction costs, because our control flow graph constructor functioned only on a subset of the code in `player`. We have since corrected the control flow graph constructor; the control flow graph construction costs reported here are precise.

13. The latter step of this simulation may somewhat understate the cost of the work that it simulates, because it postpones all set operations until the completion of the graph traversal rather than distributing them throughout the program's operation. As a result, set operations may require less time due to improved instruction cache effects. Interestingly, a recent paper by Ball [3] presents a version of our basic algorithm that, while walking the graphs, simply selects a set of edges, and then performs all set unions at the end; by reducing the number of set unions that may be required this approach improves the worst case time bound on the algorithm from $O(|E|*|E'|*|T|)$ to $O(|E|*|E'| + |E| * |T|)$ (where E and E' are the number of edges in the control flow graphs for the base and modified versions, respectively). Our simulation more accurately measures the cost of this less expensive variant of our algorithm.

We used the following procedure for this study. Given base `player` version P, modified versions $P_1 \ldots P_n$, and test suite T, we

1) used `Aristotle` to construct the control flow graphs for P,
2) ran `DejaVuSim` on P to estimate the balance of analysis costs,
3) for each version P_i, we

   a) used `Aristotle` to build the control flow graphs for P_i,
   b) inserted probes as described above to let us simulate `DejaVu`'s test selection on P and P_i,
   c) ran P_i on T, collecting test selection results in a log file,
   d) ran and validated all tests on P_i,
   e) ran and validated all selected tests on P_i.

To validate results, we again used the base version as an oracle. We compared the output of the base program (screen output and output to database files) to the corresponding outputs of the modified version. We used the Unix `time` command to measure costs, using the time for 3d as a measure of the cost of using retest-all, and summing the times for 1, 2, 3a, 3b, and 3e as a measure of the cost of performing test selection using `DejaVu`. We ran the `player` executable on the same Sun Microsystems SPARCstation 1+ used in Study 1. Due to a tendency of the client program to dump core unpredictably under SunOS, we ran the client program on a Sun Microsystems UltraSparc1/140 running Solaris 2.5.1. Our testing processes were the only active user processes on the machines. We could not similarly restrict network activity; however, we performed the study at a time when such activity was low.

3.4.4 Results

Fig. 5 depicts test selection results for Study 4. The figure contains a graph that shows the percentage of tests selected by `DejaVu` for each version of `player`. On average, over the five versions, `DejaVu` reduced test suite size over 95 percent; results varied from a reduction of 99 percent on version 2 to a reduction of 89 percent on version 3.

Table 10 shows the time savings achieved by using `DejaVu` in this study. For each version, the table shows the hours and minutes required to perform various tasks. The columns show (from left to right) the version number, the time required to run and validate all tests on the version, the time required to build control flow graphs for the base program and version and traverse them, the time required to run and validate selected tests, the time saved overall by test selection, the percentage of total time saved by test selection, and the break-even value. On average over the five modified versions, `DejaVu` reduced testing time from 7 hr and 40 min to 1 hr and 1 min; an overall reduction of 87 percent. The break-even value for the versions averaged 78; this means that in order to provide savings, `DejaVu` needs to eliminate only 78 (7.6 percent) of the 1,033 tests (assuming test costs are equal).

Of the 34–36 minutes spent on analysis in the test selection process in this study, 32 minutes were required to build control flow graphs for the base and modified versions. `DejaVuSim` required between two and four minutes to per-
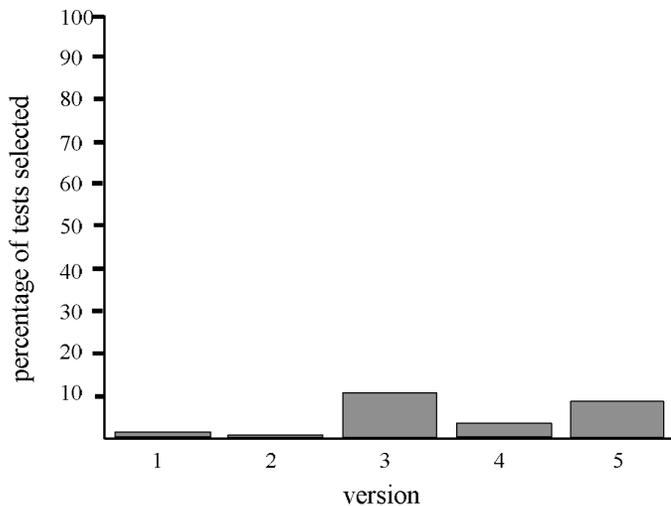
Fig. 5. Test selection statistics for Study 4.

form graph traversals and set operations. The control flow graphs for the base version could be calculated and stored during the preliminary period, trading critical period testing expense for space. On-demand construction of graphs could also reduce control flow graph construction time.

These results show that test selection *can* provide significant savings. The facts that `player` is a real, nontrivial program, representative of a significant class of programs, and that the test suite we utilized in our experimentation was a realistic, specification-based test suite of a sort we could expect to find in practice suggest that the technique may generalize to a significant class of practical regression testing situations.

5 Study 5

3.5.1 Objectives

The objective of our fifth study was to investigate the applicability of `DejaVu` to a commercial software system for which actual modified versions and tests were available.

3.5.2 Subject

For this study we obtained a commercial program, nine modified versions of the program, and the test suite that had been developed for and used to regression test the program. The program, which we call "`commercial`," is an interactive Windows NT application driven by a graphical user interface and utilized by the general end-user population. Table 11 describes the program and Table 12 shows statistics for modified versions. Note that in this case (in

contrast to our previous studies) the versions constitute a sequence of versions, each modifying the previous version; the modification information shown in Table 12 is calculated accordingly.

The test suites that were provided with `commercial` consisted of three automated scripts. Each of these scripts invoked the application, applied a number of inputs and validated their outputs, and then closed the application; in total, the three scripts applied and validated 388 inputs. We can treat these test scripts either as three tests or as 388 tests. We ultimately employed both interpretations in this study; we discuss the reasons for and ramifications of this decision later.

The `commercial` program presented some advantages not presented by the subjects used in our other studies, because although the program is relatively small, it is commercial software, and its versions and test suites were provided with the system, not constructed solely to facilitate empirical studies.

3.5.3 Procedure

The software for `commercial` contains C constructs that `Aristotle` cannot process. Thus, we could not instrument, build control flow graphs for, or run `DejaVu` on `commercial`. Whereas in Study 4 we could precisely calculate the costs of constructing control flow graphs for the programs and versions, and using these control flow graphs, we could estimate the cost of the rest of the `DejaVu` analysis, in this study we could not do this. However, by using part of the simulation procedure used in Study 4, we were able to determine the test selection results achievable by our algorithm.

To perform the simulation we inserted probes into modified versions, in a manner similar to that used in Study 4 with one exception: rather than insert code to initialize the `results` file in `main`, we initialized the file from the test scripts. Using this approach, we were able to measure test selection results when interpreting the scripts as providing either three or 388 tests. By initializing the `results` file only at the beginning of each script, we obtained results in which each script was treated as a test—for a total of three tests. By initializing the `results` file in each script prior to application and validation of each input, and after each validation saving the contents of the `results` file to a log file, we obtained results in which each input was treated as a test—for a total of 388 tests. Again, because the versions here represent a sequence of versions, our instrumentation of the $(k + 1)$th version ($k > 0$) was arranged to reveal the tests that would be selected if `DejaVu` were run on the kth and $(k + 1)$th versions.

TABLE 10
STUDY 4: EXECUTION TIMES AND SAVINGS (HR:MIN)

Version	Time to Run All Tests	Time to Perform Analysis	Time to Run Tests Selected	Time Saved	Pct. of Time Saved	Break-Even Value
1	7:43	0:34	0:05	7:02	91	76
2	7:43	0:34	0:01	7:06	92	76
3	7:40	0:36	1:08	5:58	78	80
4	7:33	0:35	0:16	6:42	89	80
5	7:42	0:36	0:40	6:28	84	78
average	7:40	0:35	0:26	6:39	87	78

TABLE 11
SUBJECT USED IN STUDY 5

Program Name	No. of Functions	Lines of Code	No. of Versions	Test Pool Size	Description of Program
commercial	27	2,145	9	3/388	Windows NT application

TABLE 12
MODIFIED VERSIONS USED IN STUDY 5

Version	Functions Modified	Lines of Code Changed
1	1	1
2	2	12
3	1	1
4	12	264
5	1	3
6	3	4
7	3	245
8	2	15
9	2	44

We used the following procedure for Study 5. Given initial version P, succession of modified versions $P_1 \ldots P_k$, and test suite T, for each modified version P_i ($i > 0$), we

1) inserted probes into the source code as just described, and
2) ran P_i, collecting test selection results in a log file.

3.5.4 Results

Fig. 6 depicts test selection statistics for this study. The graph on the left depicts results in which each script was treated as a test. The graph on the right depicts results in which each individual input in a script was treated as a test.

A comparison of these results is interesting, and reveals our motivation for employing the two interpretations of the tests. Under the three-test interpretation, DejaVu selected three tests twice, two tests twice, one test thrice, and zero tests twice, reducing the number of tests required by 48 percent overall. Under the 388-test interpretation, in all but three cases, DejaVu reduced the percentage of testing required by a greater amount than in the three-test interpretation; the overall average reduction in tests under this interpretation was 66 percent. In the other three cases—two in which no tests were selected and one in which one was selected—test selection results were the same under each in-

terpretation. These results illustrate that the "granularity" of the tests in a test suite can affect the ability of DejaVu to reduce test suite size using test selection.

An interesting additional observation is that for two of the nine modified versions considered in the study (versions 2 and 5), DejaVu selected no tests. In these two cases, there were no inputs in the test scripts that caused code modified for those versions to be executed. Thus, even if a retest-all approach were used on these programs, modified code in the versions would not be tested.

4 INTERPRETATION OF RESULTS

The studies we have described, like any other, have several threats to their validity, that we have detailed in the previous sections. Keeping these threats in mind, we draw several observations from these studies.

Studies 1, 4, and 5 illustrate that our regression test selection technique, and thus, safe test selection techniques, can reduce the cost of regression testing. Encouragingly, as Study 4 illustrates, the technique can yield greater savings when applied to larger, more complex programs than when applied to smaller, simpler programs. Furthermore, Studies 1, 2, and 3 indicate that our test selection technique can indeed achieve safety in practice. In these cases, the technique demonstrates fault-detection abilities equivalent to those of the retest-all technique.

All five studies illustrate that safe regression test selection alone is not sufficient either for reducing the cost of regression testing or for increasing the quality of that testing. As the studies illustrate, there exist programs, modified versions, and test suites for which safe test selection offers little reduction in test suite size. Characteristics of the base program, modified version, and test suite can conspire or act independently to affect test selection results. For example, as a program's structure grows more complex, the probability that an arbitrary test will execute an arbitrary modification in that program decreases. A straight-line program that contains no decision statements yields a case in which, regard-
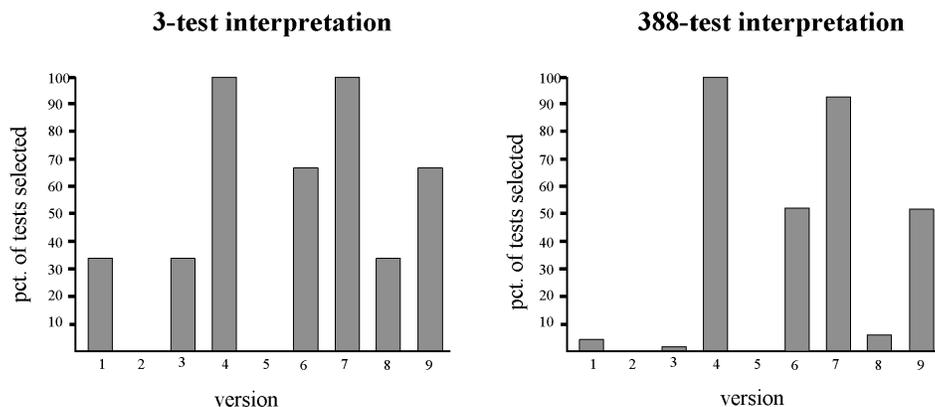

Fig. 6. Test selection statistics for Study 5.

less of the nature of modifications and test suites, our test selection technique always selects all tests in *T*. As a second example, in an arbitrary program, a code modification that is conditionally executed is less likely to be reached by an arbitrary test than a code modification that is necessarily executed. For the version of our test selection technique utilized in these studies, a modification in any statement of *P* that precedes all branches in *P* necessitates selection of all tests through *P*, regardless of the structure of *P* or the nature of the tests in *T*. As a third example, if a given test executes a large percentage of the statements in an arbitrary program, the probability that the test will be selected for an arbitrary modified version of that program is greater than the probability that a test that executes a small percentage of the statements in the modified program will be selected. If a test executes every statement in *P*, then regardless of the structure of *P* and the location of modifications in *P*, our test selection technique will select that test.

Even when test selection reduces the number of tests that must be executed, the analysis required to select tests can outweigh reductions in test execution and validation costs. As an extreme example, in our second and third studies, analysis time always outweighed the time required to reexecute all tests. Thus, for certain programs and test suites, test selection cannot be cost-effective. This result does not imply that test selection cannot be cost-effective for small programs; rather, the result reflects the fact that cost-effectiveness depends on *both* the cost of analysis *and* the cost of executing tests. The potential for regression test selection to provide savings for a particular software system and its test suite must therefore be assessed in terms of both of these factors.

Given the observed potential for variations in test selection results, we would like to find plausible predictors that can tell us in advance, with a reasonable degree of certainty, whether or not test selection is likely to be effective. Toward this end, Rosenblum and Weyuker [30] propose *coverage-based predictors* for use in predicting the cost-effectiveness of selective regression testing strategies. Their predictors use the average percentage of test cases that execute covered *entities*—such as statements, branches, or functions—to predict the number of tests that will be selected when a change is made to those entities. Subsequent empirical studies by Harrold, Rosenblum, Rothermel and Weyuker [13] show that, although the Rosenblum-Weyuker predictor is relatively effective at predicting the *average* effectiveness of a regression test selection strategy, it may significantly under- or overestimate test selection results for particular versions. The authors show how to improve the predictive power of the Rosenblum-Weyuker predictor both generally and for specific versions by incorporating information on the distribution of modifications.

Even when regression test selection is effective, it may not be adequate. Our fifth study, in particular, demonstrates this, in the cases where no tests through modified code were found. A regression test selection technique selects a subset of an existing test suite; given a program that is not adequately tested by its existing test suite, it is not likely that any subset of that test suite will be adequate for a modified version of that program. Furthermore, even if the

previous version of a program had been adequately tested by its existing test suite, the tests in that suite may no longer be adequate for the modified program. Regression test selection techniques, as we have defined them, address only the problem of selecting tests from an existing test suite; safe techniques promise only to not omit tests, from the original test suite, that would reveal faults if executed. A complete regression testing effort should look beyond existing test suites. Many of the selective retest techniques cited in Section 2 of this paper consider this regression testing adequacy problem in addition to the problem of test selection; however, to our knowledge, with the exception of recent work by Binkley [6], these techniques have not been empirically investigated.

Data calculated from the results of Studies 2 and 3 support a hypothesis that for purposes of regression test selection, code-coverage-based test suites may be preferable to non-coverage-based test suites. This support comes in two forms. First, in cases where selection of a large percentage or a small percentage of tests predominates, non-coverage-based test suites display a greater tendency than coverage-based test suites to require selection of almost all, or very few, of the tests in the suites, respectively. In the former case, the chance that test selection can be cost-effective is reduced; in the latter case, the chance that test selection can locate tests through modified code (and locate errors in that code) is reduced. Second, in cases where test selection selects between 20 and 80 percent of the test suite, non-coverage-based test suites are associated with greater variance in test selection results than coverage-based test suites. This fact may make accurate prediction of test selection results for a particular testing load more difficult for non-coverage-based test suites than for coverage-based suites. Moreover, our intuition suggests that the amount of retesting required for a given modification should be more closely related to the modification than to the test suite, but noncoverage suites accentuate the role of the test suite in test selection.

Finally, as the results of Study 5 in particular illustrate, test granularity is an important factor in determining the success of test selection. Finer granularity tests increase the likelihood of achieving efficiency gains with regression test selection, by decreasing the likelihood (on average) that tests will execute modified code. However, these potential gains must be balanced against potential losses: finer granularity tests may require greater overhead than coarser granularity tests. For example, in Study 5, the finest possible test granularity could require us to open an application 388 times, as opposed to three times, significantly increasing overall testing time. Furthermore, finer granularity tests may decrease the opportunity to identify faults that arise due to code interactions.

5 CONCLUSIONS

In this paper, we have presented the results of several empirical studies of a safe regression test selection technique. Our results indicate that safe regression test selection can be cost-effective, and that in practice it can indeed avoid omitting tests that would not be omitted by a retest-all approach.

The results also indicate, however, that the cost-effectiveness of test selection can vary widely based on a number of factors. In particular, the cost of analysis necessary for test selection and the cost of executing and validating tests interact to affect cost-effectiveness. By calculating break-even values we may be able to determine, for a given software product and test suite, whether test selection can ever be cost-effective for that product and test suite. In cases where break-even values indicate that test selection may be cost-effective, however, that cost-effectiveness is still not guaranteed: it depends on factors of the test suite, such as code-coverage characteristics and test granularity, and also on locations of modifications.

Further experimentation with a wider range of programs, versions, and test suites is necessary, to determine the extent to which, and conditions under which, these results may generalize to industrial practice.

Given the dependence of test selection effectiveness on test suite characteristics and code modification patterns, our results also suggest that in practice, it may be possible to design test suites that promote the use of regression test selection techniques. If such test suites can be constructed without loss to other aspects of test adequacy, design for regression testability may be appropriate and possible. It may be similarly possible to specify code modification practices that promote the use of test selection. We are currently investigating these issues.

In addition to experiments with individual test selection algorithms, comparative experiments of various algorithms are also mandated. To date, only two such experiments have been reported in the literature [11], [29]. Such experiments should examine the relative cost-benefits of different families of test selection algorithms: for example, minimization versus safe algorithms. Such experiments should also examine the relative cost-benefits of algorithms within the same family, such as the various safe algorithms, or the various versions of our graph-walk-based algorithms. The work we report here provides an infrastructure for such further experimentation, by ourselves and by other researchers.

ACKNOWLEDGMENTS

REFERENCES

[1] H. Agrawal, J. Horgan, E. Krauser, and S. London, "Incremental Regression Testing," *Proc. Conf. Software Maintenance*, pp. 348–357, Sept. 1993.
[2] M. Balcer, W. Hasling, and T. Ostrand, "Automatic Generation of Test Scripts from Formal Test Specifications," *Proc. Third Symp. Software Testing, Analysis, and Verification*, pp. 210–218, Dec. 1989.
[3] T. Ball, "On the Limit of Control Flow Analysis for Regression Test Selection," *Proc. Int'l Symp. Software Testing and Analysis, ISSTA*, Mar. 1998.
[4] S. Bates and S. Horwitz, "Incremental Program Testing Using Program Dependence Graphs," *Proc. 20th ACM Symp. Principles of Programming Languages*, Jan. 1993.
[5] P. Benedusi, A. Cimitile, and U. De Carlini, "Post-Maintenance Testing Based on Path Change Analysis," *Proc. Conf. Software Maintenance*, pp. 352–361, Oct. 1988.
[6] D. Binkely, "Semantics Guided Regression Test Cost Reduction," *IEEE Trans. Software Eng.*, vol. 23, no. 8, Aug. 1997.
[7] D. Binkley, "Reducing the Cost of Regression Testing by Semantics Guided Test Case Selection," *Proc. Conf. Software Maintenance*, Oct. 1995.
[8] Y.F. Chen, D.S. Rosenblum, and K.P. Vo, "TestTube: A System for Selective Regression Testing," *Proc. 16th Int'l Conf. Software Eng.*, pp. 211–222, May 1994.
[9] K.F. Fischer, "A Test Case Selection Method for the Validation of Software Maintenance Modification," *Proc. COMPSAC'77*, pp. 421–426, Nov. 1977.
[10] K.F. Fischer, F. Raji, and A. Chruscicki, "A Methodology for Retesting Modified Software," *Proc. Nat'l Telecommunications Conf. B-6-3*, pp. 1–6, Nov. 1981.
[11] T.L. Graves, M.J. Harrold, J-M. Kim, A. Porter, and G. Rothermel, "An Empirical Study of Regression Test Selection Techniques," *Proc. 20th Int'l Conf. Software Eng.*, Apr. 1998.
[12] R. Gupta, M.J. Harrold, and M.L. Soffa, "An Approach to Regression Testing Using Slicing," *Proc. Conf. Software Maintenance*, pp. 299–308, Nov. 1992.
[13] M.J. Harrold, D.S. Rosenblum, G. Rothermel, and E.J. Weyuker, "Empirical Studies of a Prediction Model for Regression Test Selection," Technical Report OSU-CISRC-2/98-TR55, Ohio State Univ., Feb. 1998.
[14] M.J. Harrold and G. Rothermel, "Aristotle: A System for Research on and Development of Program-Analysis-Based Tools," Technical Report OSU-CISRC-3/97-TR17, Ohio State Univ., Mar. 1997.
[15] M.J. Harrold and M.L. Soffa, "An Incremental Approach to Unit Testing During Maintenance," *Proc. Conf. Software Maintenance*, pp. 362–367, Oct. 1988.
[16] M.J. Harrold and M.L. Soffa, "An Incremental Data Flow Testing Tool," *Proc. Sixth Int'l Conf. Testing Computer Software*, May 1989.
[17] J. Hartmann and D.J. Robson, "RETEXT—Development of a Selective Revalidation Prototype Environment for Use in Software Maintenance," *Proc. 23rd Hawaii Int'l Conf. System Sciences*, pp. 92–101, Jan. 1990.
[18] J. Hartmann and D.J. Robson, Techniques for Selective Revalidationk" *IEEE Software*, vol. 16, no. 1, pp. 31–38, Jan. 1990.
[19] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the Effectiveness of Dataflow and Controlflow-based Test Adequacy Criteria," *Proc. 16th Int'l Conf. Software Eng.*, pp. 191–200, May 1994.
[20] R. Johnson, *Elementary Statistics*, sixth edition. Belmont, Calif.: Duxbury Press, 1992.
[21] J. Laski and W. Szermer, "Identification of Program Modifications and Its Applications in Software Maintenance," *Proc. Conf. Software Maintenance*, pp. 282–290, Nov. 1992.
[22] J.A.N. Lee and X. He, "A Methodology for Test Selection," *J. Systems and Software*, vol. 13, no. 1, pp. 177–185, Sept. 1990.
[23] H.K.N. Leung and L. White, "Insights into Regression Testing," *Proc. Conf. Software Maintenance*, pp. 60–69, Oct. 1989.
[24] H.K.N. Leung and L. White, "Insights into Testing and Regression Testing Global Variables," *J. Software Maintenance*, vol. 2, pp. 209–222, Dec. 1990.
[25] H.K.N. Leung and L.J. White, "A Study of Integration Testing and Software Regression at the Integration Level," *Proc. Conf. Software Maintenance*, pp. 290–300, Nov. 1990.
[26] H.K.N. Leung and L.J. White, "A Cost Model to Compare Regression Test Strategies," *Proc. Conf. Software Maintenance*, pp. 201–208, Oct. 1991.
[27] T.J. Ostrand and M.J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests," *Comm. ACM*, vol. 31, no. 6, June 1988.
[28] T.J. Ostrand and E.J. Weyuker, "Using Dataflow Analysis for Regression Testing," *Proc. Sixth Ann. Pacific Northwest Software Quality Conf.*, pp. 233–247, Sept. 1988.

[29] D.S. Rosenblum and G. Rothermel, "An Empirical Comparison of Regression Test Selection Techniques," *Proc. Int'l Workshop for Empirical Studies of Software Maintenance*, pp. 89–94, Oct. 1997.

[30] D.S. Rosenblum and E.J. Weyuker, "Using Coverage Information to Predict the Cost-Effectiveness of Regression Testing Strategies," *IEEE Trans. Software Eng.,* vol. 23, no. 3, pp. 146–156, Mar. 1997.

[31] G. Rothermal, "Efficient, Effective Reqression Testing Using Safe Test Selection Techniques," Technical Report 96–101, Clemson Univ., Jan. 1996.

[32] G. Rothermel and M.J. Harrold, "A Safe, Efficient Algorithm for Requression Test Selection," *Proc. Conf. Software Maintenance*, pp. 358–367, Sept. 1994.

[33] G. Rothermel and M.J. Harrold, "Analyzing Reqression Test Selection Techniques," *IEEE Trans. Software Eng.*, vol. 22, no. 8, pp. 529–551, Aug. 1996.

[34] GT. Rothermel and M.J. Harrold, "A Safe, Efficient Regression Test Selection Technique," *ACM Trans. Software Eng. and Methdology*, vol. 6, no. 2, pp. 173–210, Apr. 1997.

[35] B. Eherlund and B. Korel, "Modification Oriented Software Testing," *Conf. Proc.: Quality Week*, pp. 1–17, 1991.

[36] B. Sherlund and B. Korel, "Logical Modification Oriented Software Testing," *Proc. 12th Int'l Conf. Testing Computer Software*, June 1995.

[37] A.B. Taha, S.M. Thebaut, and S.S. Liu, An Approach to Software Fault Localization and Revalidation Based on Incremental Data Flow Analysis," *Proc. 13th Ann. Int'l Computer Software and Applications Conf.*, pp. 527–534, Sept. 1989.

[38] F. Vokolos and P. Frankl, "Pythia: A Regression Test Selection Tool Based on Textual Differencing," *Proc. Third Int'l Conf Reliability, Quality, and Safety of Software Intensive Systems, ENCRESS'97*, May 1997.

[39] L.J. Wshite and H.K.N. Leung, "A Firewall Concept for Both Control-Flow and Data-Flow in Regression Integration Testing," *Proc. Conf. Software Maintenance*, pp. 262–270, Nov. 1992.

[40] L.J. White, V. Narayanswamy, T. Friedman, M. Kirschenbaum, P. Piwowarski, and M. Oha, "Test Manager, A Regression Testing Tool," *Proc. Conf. Software Maintenance*, pp. 338–347, Sept. 1993.

[41] S.S. Yau and Z. Kishimoto, "A Method for Revalidating Modified Programs in the Maintenance Phase," *COMPSAC'87: Proc. 11th Ann. Int'l Computer Software and Applications Conf.,* pp. 272–277, Oct. 1987.

Gregg Rothermel received the BA degree in philosophy from Reed College; an MS degree in computer science from SUNY Albany; and the PhD degree in computer science from Clemson University. He is currently an assistant professor in the Computer Science Department of Oregon State University. His research interests include software engineering and program analysis, with an emphasis on the application of program analysis techniques to problems in software maintenance and testing. He is a recipient of the National Science Foundation's Faculty Early Career Development Award. Previous positions include vice president, quality assurance and quality control, Palette Systems, Incorporated. Dr. Rothermel is a member of the IEEE and the ACM.

Mary Jean Harrold received the MS and MA degrees in mathematics from Marshall University, and the MS and PhD degrees in computer science from the University of Pittsburgh. She is currently an associate professor in the Department of Computer and Information Science at Ohio State University, where she leads the Aristotle Research Group. Her research interests include program analysis and testing, and testing of object-oriented software. She is a recipient of the National Science Foundation's National Young Investigator Award. Dr. Harrold is a member of the IEEE, and the ACM.