

An Empirical Comparison of Two Safe Regression Test Selection Techniques

Phyllis G. Frankl
Computer and Information
Science Department
Polytechnic University
Brooklyn, NY 11201
pfrankl@poly.edu

Gregg Rothermel, Kent Sayre
Department of
Computer Science
Oregon State University
Corvallis, OR 97331
grother@cs.orst.edu

Filippos I. Vokolos
Department of
Computer Science
Drexel University
Philadelphia, PA 19104
filip@drexel.edu

Abstract

Regression test selection techniques reduce the cost of regression testing by selecting a subset of an existing test suite to use in retesting a modified program. Safe regression test selection techniques guarantee (under specific conditions) that the selected subset will not omit faults that could have been revealed by the entire suite. Many regression test selection techniques have been described in the literature. Empirical studies of some of these techniques have shown that they can be beneficial, but only a few studies have empirically compared different techniques, and fewer still have considered safe techniques. In this paper, we report the results of a comparative empirical study of implementations of two safe regression test selection techniques: DeJaVu and Pythia. Our results show that, despite differences in their approaches, and despite the theoretically greater ability of DeJaVu to select smaller test suites than Pythia, the two techniques often selected equivalent test suites in practice, at comparable costs. These results suggest that factors such as ease of implementation, generality, and availability of supporting tools and data may play a greater role than cost-effectiveness for practitioners choosing between these techniques.

1 Introduction

Regression testing is the process of testing modified software to detect whether new faults have been introduced into previously tested code. One approach to regression testing is to save the test suites for a product and reuse them to retest the product after it is modified, but this *retest-all* technique can be excessively expensive. *Regression test selection* (RTS) techniques (e.g. [5, 19, 23]), in contrast, attempt to reduce the cost of regression testing by selecting a subset of the existing test suite to use on the modified program.

Most RTS techniques are not *safe*. Techniques that are not safe can fail to select test cases that reveal faults in the modified program. In this paper, we focus on safe RTS techniques. When an explicit set of safety conditions is satisfied, safe RTS techniques select all test cases, *in the original test suite*, that can reveal faults in the modified program [18]. Of course, under these same safety conditions the retest-all technique is safe, but it requires execution of all test cases. Key to the cost-effectiveness of safe techniques, then, is *precision*: the ability of a technique to exclude unnecessary test cases.

In this paper, we present the results of a comparative empirical study of two safe RTS techniques. The techniques we studied have been implemented as the tools DeJaVu [19] and Pythia [23]. These techniques present tradeoffs of the sort that implementors and users of safe RTS techniques must consider. DeJaVu examines source code at the level of individual statements, and is among the most (theoretically) precise techniques available. Pythia employs a theoretically less precise analysis, examining modified code at the statement level but selecting test cases at the level of enclosing basic blocks. The two approaches differ in terms of ease of implementation, implementation requirements (supporting tools and data required) and generality (ease of application to other languages and systems).

To examine the effects of these tradeoffs between Pythia and DeJaVu, we performed an experiment, in which we applied both techniques to a set of programs and modified versions of those programs, on large sets of test suites of two different types. Our results showed that, despite the differences in their approaches, and the theoretically greater ability of DeJaVu to select smaller test suites, the two techniques often selected equivalent test suites in practice, at comparable costs. These results suggest that factors such as ease of implementation, generality, and availability of supporting tools and data may

play a greater role than cost-effectiveness for practitioners choosing between these techniques.

In the next section, we review the relevant literature, and describe the algorithms underlying, and implementations of, `DejaVu` and `Pythia`. In Section 3, we present our experiment design and results. Section 4 provides additional discussion of those results, and Section 5 concludes.

2 Background and Related Work

In the following discussion, let P be a program, let P' be a modified version of P , and let S and S' be the specifications for P and P' , respectively. Let T be a test suite developed initially for P .

2.1 Safe Regression Test Selection

Regression testing involves several subproblems [13, 18], including the problems of selecting a subset of the original test suite to run, generating new test cases to exercise new code or functionality, executing old and new test cases, and constructing a new test suite for use on future versions. While each of these problems is important, in this work we focus on the regression test selection problem: the problem of selecting a subset of an existing test suite T to execute on P' . Regression test selection (RTS) techniques (e.g. [5, 19, 23, 27]) address this problem.

In this paper we are primarily interested in *safe* RTS techniques ([2, 5, 12, 19, 23]), which by definition, given that certain preconditions are met, eliminate only test cases that are provably not able to reveal faults. Informally, these preconditions require that: (1) the test cases in T are expected to produce the same outputs on P' as they did on P ; i.e., the specifications for these test cases have not changed; and (2) test cases can be executed deterministically, holding all factors that might influence test behavior constant with respect to their states when P was tested with T . (This notion of safety is defined formally, and these preconditions are expressed more precisely, in [18].) The techniques we examine in this paper, implemented as `DejaVu` and `Pythia`, are both safe given the same set of preconditions.

Reference [18] presents a framework for analyzing and comparing RTS techniques, part of which we employ in this work. The framework consists of four criteria: inclusiveness, precision, efficiency, and generality. *Inclusiveness* measures the extent to which a technique selects test cases from T that can expose differences between P and P' — for the safe techniques that we consider, 100% inclusiveness is required. *Precision* measures the ability of a technique to omit unnecessary test

cases. *Efficiency* measures the time and space requirements of a technique. *Generality* measures the range of situations (including types of systems, programs and languages accommodated) in which a technique is capable of functioning.

In assessing efficiency, we distinguish between costs incurred in the *preliminary phase* of regression testing and costs incurred in the *critical phase*. The *critical phase* of regression testing occurs following completion of modifications creating P' from P , and prior to release of P' , and is the (typically limited) time during which regression testing must be accomplished. Preliminary phase costs are incurred prior to the critical period, incrementally, as the project develops. For instance, source code control and configuration systems can incrementally maintain information used later by the analysis portion of a safe RTS technique, and program profiling information for P can be gathered, following its release, and prior to the critical period. Techniques can take advantage of these phases, scheduling activities in the preliminary phase where possible. In our analyses, we focus on critical phase costs.

2.2 Two Safe Techniques

2.2.1 `DejaVu`

Rothermel and Harrold developed a family of safe RTS techniques, one of which is implemented as the tool `DejaVu`. We provide an overview of `DejaVu` here; reference [19] presents the detailed algorithm, with cost analyses and examples of its use.

`DejaVu` operates on C programs, constructing control-flow graph (CFG) representations of the procedures in P and P' , in which individual nodes are labeled by their corresponding statements. `DejaVu` assumes that a *test history* is available that records, for each test case t in T and each edge e in the CFG for P , whether t traversed e . This test history is gathered by instrumentation code inserted into the source code of the system under test. Construction and storage of the CFGs for the base program P and the test history are preliminary phase costs, since they can be accomplished during off-peak times before the critical phase. Construction of the CFGs for P' , however, is a critical-phase cost.

`DejaVu` performs a simultaneous depth-first graph walk on a pair of CFGs G and G' for each procedure and its modified version in P and P' , keeping pointers $\uparrow N$ and $\uparrow N'$ to the current node in each graph. The tool begins with the entry nodes of G and G' ; it then executes a recursive depth-first search on both CFGs. Upon visiting a pair of nodes N and N' in G and G' , the tool examines each edge e leaving N to determine whether there is an

equivalently labeled edge in G' . If not, the tool places e into a set of *dangerous edges*. If there is a corresponding edge in G' , and that edge enters an already traversed portion of the graph at the same node in both CFGs, then the current recursion is terminated. Otherwise, $\uparrow N$ and $\uparrow N'$ are moved forward to point to a new pair of nodes. The tool then checks to see if the labels on $\uparrow N$ and $\uparrow N'$ are lexically equivalent (textually equivalent after white space and comments have been removed). If they are not, it places the edge just followed into the set of dangerous edges and returns to the source of that edge, ending that trail of recursion.

After DeJaVu has determined the dangerous edges that it can reach by crossing non-dangerous edges, it terminates. At this point, any test case $t \in T$ is selected for retesting P' if the execution trace for t contains a dangerous edge.

DeJaVu requires input from a control flow graph construction tool and an instrumenter that functions at the level of control flow graph nodes, and such tools are relatively expensive to implement. On the other hand, the algorithm underlying DeJaVu applies, with minor extensions, to procedural-language programs generally, and the tool can be applied to any systems for which the required control flow graph model and profiling data are provided in proper format. (To date the algorithm and tool have been extended to handle systems built in C++ [21], Java [7] and Ada-83).

2.2.2 Pythia

Vokolos and Frankl developed a *textual differencing* approach to safe regression test selection, and implemented this approach as the tool Pythia. What follows in this section provides an overview of Pythia; for a more detailed description refer to [23, 25].

Pythia operates on C programs. The tool was implemented by integrating three UNIX programs: `cc`, the C language compiler; `pretty`, a beautifier for C programs; and `diff`, the general purpose file comparison program. Given P , P' , and T , to select a regression test suite T' from T , Pythia maintains a history of the basic blocks executed by each test case in T , and compares the source files of P and P' to identify modified statements. Each modified statement is mapped to the appropriate basic block and all test cases that executed that basic block are selected. The source-to-source comparison is done with `diff`. Since `diff` treats each line simply as a character string, it could report differences with no lexical or syntactic import, such as formatting differences. To avoid this, Pythia first transforms each program version into a canonical form. The `prettyprinter`,

`pretty` [15, 22], which transforms code into an aesthetically appropriate format with uniform conventions on indentation and line breaks, is used to accomplish this. Execution tracing is accomplished by instrumenting the source code using the C compiler.

One difference between Pythia and DeJaVu involves the granularity at which they perform their analyses. Pythia associates differences in source code lines with the basic blocks in P whose behavior could change as a result, then selects all test cases that executed the selected basic blocks. In the simplest situation, a change is contained within a single basic block; in such cases, Pythia selects all test cases that execute that block. In more complicated situations, a textual difference may change the basic block structure of the program. In such situations, Pythia selects a basic block b (possibly at a shallower depth of nesting) such that all test cases that may be affected by the code modifications execute block b . For example, under some circumstances, changes to the `then` clause in an `if` statement may cause selection of all test cases that reach the `if` condition, regardless of whether they cause it to evaluate to “true” or “false”.

Given this procedure, Pythia is (at least theoretically) less precise than DeJaVu, potentially selecting larger test suites. Pythia, however, can be implemented from common UNIX utilities, making its implementation for C programs simpler than that of DeJaVu, albeit at reduced generality.

2.3 Related Work

RTS techniques have been evaluated and compared analytically in [18], but only recently have attempts been made to evaluate or compare them empirically [3, 4, 5, 6, 8, 11, 17, 19, 20, 24, 26]. Of these studies, most focus on single techniques. In contrast, the work described in this paper focuses on safe RTS techniques and on comparative empirical analysis.

Three recent studies have focused on comparisons of techniques. In [6, 11], three types of techniques, including a safe technique, a minimization technique, and another non-safe technique, were empirically compared to random test selection for relative precision and inclusiveness. In [3], results of a study comparing DeJaVu to the safe RTS technique `TestTube` [5] were reported, in which the two were compared for relative precision on two subject groups of programs. The study we report in this paper is also comparative, and like the study in [3], focuses on safe techniques; however, this study examines techniques not previously empirically compared.

3 Experiment

Our primary objective was to empirically investigate the relative cost-benefit tradeoffs involved in utilizing `Pythia` or `DeJaVu` for regression test selection. We also wished to compare the cost-benefits of using those tools to those of not using regression test selection at all, and to determine whether these tradeoffs depend on the type of test suite being used.

3.1 Measures

To meet these objectives, we consider the percentages of test cases selected by `DeJaVu` and `Pythia` over various programs and test suites. In terms of the comparison framework described in Section 2, these percentages constitute the *precisions* of the techniques. We measure precision as $(|T'|/|T|) * 100$, where $|T|$ and $|T'|$ are the sizes of the initial and selected test suite, respectively.

Note that under this measure, *lower* precision numbers, representing lower percentages of test cases selected, are better, indicating greater savings. Further, since savings in the cost of testing are proportional to the cost of executing and validating test cases [14], precision provides a way to measure and compare relative savings. The percentages may then be compared to the cost, in terms of test suite size, of retest-all (100%).

3.2 Experiment Instrumentation

3.2.1 Subject Programs, Versions, and Tests

We used eight C programs as subjects. The first seven programs, with faulty versions and test cases, were assembled by researchers at Siemens Corporate Research for a study of the fault-detection capabilities of control-flow and data-flow coverage criteria [9]. We refer to these as the *Siemens programs*. The eighth program, `space`, is a program developed for the European Space Agency. We refer to this program as the *Space program*. Table 1 describes the programs. The table lists, for each program, the number of lines of executable code in that program, the number of modified versions available, the size of the initial test pool, and the average size of its test suites. (The information in these last three columns is further described below.)

The Siemens programs. The Siemens programs perform various tasks: `tcas` implements a collision avoidance algorithm, `schedule1` and `schedule2` are priority schedulers, `totinfo` computes statistics, `printtok1` and `printtok2` are lexical analyzers, `replace` performs pattern matching and substitution. For each program, the Siemens researchers created a

Program	Lines of Code	Versions	Test Pool Size	Test Suite Avg. Size
<code>tcas</code>	138	41	1608	6
<code>schedule1</code>	299	9	2650	8
<code>schedule2</code>	297	10	2710	8
<code>totinfo</code>	346	23	1052	7
<code>printtok1</code>	402	7	4130	16
<code>printtok2</code>	483	10	4115	12
<code>replace</code>	516	32	5542	19
<code>space</code>	6218	35	13585	155

Table 1. Experiment subjects.

test pool of black-box test cases using the category partition method and TSL tool [1, 16]. (These test pools serve as universes of potential test cases, and can be used as a source for multiple test suites.) They then augmented this test pool with manually created white-box test cases to ensure that each exercisable statement, edge, and definition-use pair in the base program or its control flow graph was exercised by at least 30 test cases. The researchers also created faulty versions of each program by modifying code in the base version; in most cases they modified a single line of code, and in a few cases they modified between 2 and 5 lines of code. Their goal was to introduce faults that were as realistic as possible, based on their experience with real programs. To obtain meaningful results, the researchers retained only faults that were detectable by at least 3 and at most 350 test cases in the associated test pool.

Space program. The Space program is an interpreter for the Array Definition Language (ADL), an application-specific language which allows the user to describe the configuration of an array of antennas. The program reads a file of ADL statements, and checks the contents of the file for adherence to the ADL grammar and specific consistency rules. If the ADL file is correct, the Space program outputs a data file containing a list of antenna elements, positions, and excitations; otherwise the program outputs error messages. The Space program has 35 versions, each containing a single fault: 30 of these were discovered during the program’s development; we discovered five others. The test pool for the Space program was constructed in two phases. We began with a pool of 10,000 randomly generated test cases. Then we added new test cases until every executable edge in the program’s control flow graph was exercised by at least 30 test cases. This process yielded a test pool of 13,585 test cases.

Test Suites. To obtain sample test suites for these programs, we used the test pools for the base programs and

test-coverage information about the test cases in those pools to create 500 branch-coverage-adequate test suites for each program. Each suite was created by randomly selecting (without replacement) test cases from the appropriate pool, adding a given test case to the suite if it adds coverage, and continuing until coverage was 100% branch-coverage-adequate. We also created a second set of test suites for each program, by randomly selecting (without replacement), from the program's test pool, test suites of sizes equivalent to the branch-adequate suites. In the sequel, we refer to these two varieties of test suites as *branch-coverage* and *random* test suites, respectively.

3.2.2 Instrumentation

To perform the experiment we used existing implementations of the *Pythia* and *DejaVu* algorithms. The *DejaVu* tool, described in [19], implements the basic technique outlined in Section 2.2.1. The *Pythia* tool, described in [23], implements the textual differencing technique outlined in Section 2.2.2. Test execution was automated by encoding test input and output validation instructions in Unix shell scripts.

3.3 Experiment Design

The experiment was run using an $8 \times 2 \times 2$ factorial design with 500 precision measures per cell; the three categorical factors were:

- The subject program (8 programs, each with a variety of modified versions).
- The RTS technique (*Pythia* and *DejaVu*).
- The test suite type (branch-coverage and random).

For each subject program P with each of its versions, for each test suite type, the RTS techniques *Pythia* and *DejaVu* were applied to each of the 500 sample test suites of that test suite type. The sizes of the selected test suites were collected and used in calculating precision values. These precision values were used as the statistical data set.

3.4 Threats to Validity

In this section we discuss the potential threats to validity for this experiment.

3.4.1 External Validity

The threats to external validity for our experiment are centered around the issue of how representative the subject programs studied are. The Siemens programs, although nontrivial, are small, and larger programs may be subject to different cost-benefit tradeoffs. *Space*

is a larger program; however, it is only one such program. Further, peculiarities such as the complexity of the control- and data-flow in the code of the subject programs, the kind and locality of changes to the subject programs, and the composition of the subject test suite all contribute threats to the external validity of the results. To reduce these threats, we used a factorial design to apply each RTS tool to each test suite and subject program.

The fact that the modifications in the Siemens programs involved synthetic (seeded) faults, some of which are similar and some of which affect the same program statements, may also affect our ability to generalize our results. The faults in *space* were not artificially injected; still, we cannot claim that these faults are representative of typical faults encountered during software development across programs generally, and thus, the modifications made to correct the faults may not be representative. In all versions of the programs, however, the amount of modified code is small; results may differ when larger and more complex patterns of modification are present.

Finally, our experiment was performed using branch-coverage-adequate and random test suites. The branch-coverage-adequate suites utilized do represent one type of test suite that might be used in practice; however, it is possible that *DejaVu* and *Pythia* will compare differently when used with other types of test suites.

In general, such threats to external validity can be addressed only by additional studies on additional subjects.

3.4.2 Internal Validity

Our greatest concern with respect to internal validity in these studies are instrumentation effects that can bias our results. One source of such effects are faults in the tools and scripts utilized. To reduce the likelihood of such effects, we performed code reviews on our tools, and validated tool outputs on several small programs.

3.4.3 Construct Validity

In these studies, our measurement of precision partially captures the factors important to assessing the cost-benefits of regression test selection. However, this measure does not account for the possibility that test cases may have different costs. Our measure also makes several simplifying assumptions. It assumes that the cost of executing test cases is the same under regression test selection and retest-all, and that all test cases have uniform costs. However, the measure has proven to be an accurate reflection of corresponding savings in regression testing time in previous studies [20], and has the additional advantage of simplicity.

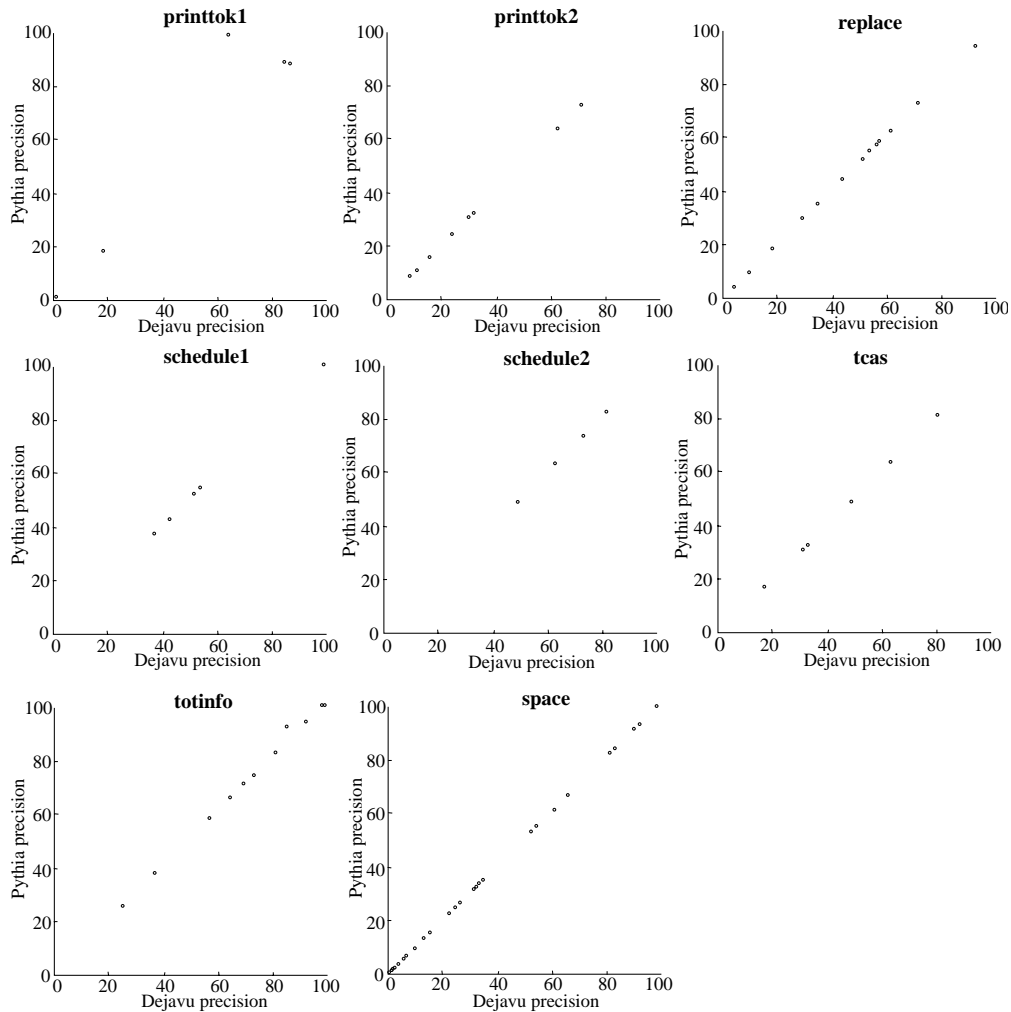


Figure 1. Precision data for DeJaVu and Pythia on branch-coverage-adequate test suites.

3.4.4 Conclusion Validity

Although we utilized a large number of test suites in this study, the number of programs and versions used might not have been large enough to detect other patterns. The use of additional versions might have created an opportunity for finding significant differences in precision.

3.5 Results and Analysis

Figure 1 depicts the precision values measured in the experiment, for branch-coverage-adequate test suites. The figure contains a graph for each of the eight subject programs. In each graph, each point represents the average percentage of test cases selected on a particular version of that program: the position of the point along the horizontal axis indicates the precision of DeJaVu, on average over the 500 test suites used for that version;

the position of the point along the horizontal axis indicates the precision of Pythia, on average over the 500 test suites for that version. (Note that some overplotting has occurred, on versions in which selection results are equivalent.) Points on the $x = y$ line indicate versions on which DeJaVu and Pythia selected, on average over the test suites, the same number of test cases.

As the figure shows, both techniques frequently reduce test suite size, often by more than 50%. (Recall that lower precisions are better than higher precisions.) Also, a larger number of cases in which precision was gained were observed on the larger Space program than on the Siemens programs, suggesting that the opportunity for savings may increase with larger programs. Precision results do vary, however, across versions; on some versions of several programs, including

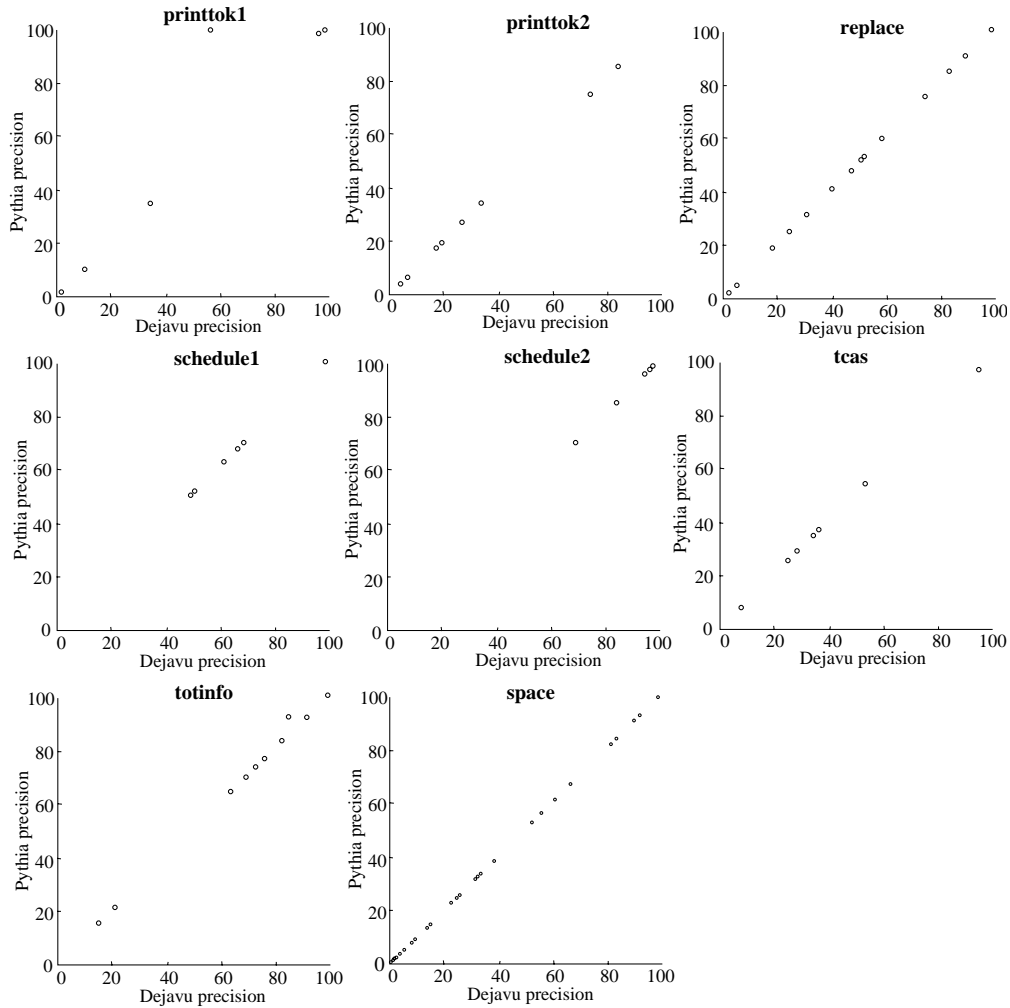


Figure 2. Precision data for DeJaVu and Pythia on random test suites.

some versions of *Space*, no reduction in test suite size was observed. Where *DeJaVu* is concerned, these precision results have been observed previously and are discussed in [20], and the reader can refer to that discussion for details. Where *Pythia* is concerned, similar results have been observed on other programs, including *Space* [24], though not the Siemens programs.

Where these results present new insights is with respect to the comparison of the two RTS techniques. As Figure 1 shows, for branch-coverage-adequate test suites, on the Siemens programs the two techniques were almost always equally precise, and on *Space*, they were always equally precise. Altogether, there were only four modified program versions on which the two techniques differed in their test selection: in these four cases *DeJaVu* was more precise than *Pythia*. Specif-

ically (on average over the test suites), on one version of *printtok1* *Pythia* selected 100.0% of the test cases whereas *DeJaVu* selected 63.8%, and on a second version of *printtok1* *Pythia* selected 87.6% of the test cases whereas *DeJaVu* selected 86.9%. Also, on one version of *totinfo* *Pythia* selected 92.6% of the test cases whereas *DeJaVu* selected 85.3%, and on a second version of *totinfo* *Pythia* selected 100.0% of the test cases whereas *DeJaVu* selected 99.5%.

Results on random test suites (Figure 2) are similar to those observed on branch-coverage-adequate test suites. Both techniques exhibited identical precision on all but five versions. These included the same two versions of *printtok1*, with (on average over the test suites) *Pythia* selecting 100.0% and 97.4% of the test cases, and *DeJaVu* selecting 53.3% and 96.0% of the

test cases, respectively, and the same two versions of `totinfo`, with `Pythia` selecting 91.8% and 100.0% percent of the test cases, and `DejaVu` selecting 85.3% and 99.8% of the test cases, respectively. A small difference was also observed on `tcas`, with `Pythia` selecting 8.1% of the test cases and `DejaVu` selecting 8.0% of the test cases.

We also compared the four testing strategies (i.e., the four combinations of RTS technique and type of test suite) cumulatively for all 170 versions of the eight subject programs. We considered four variables: `Pythia-cov`, the precision of `Pythia` on the branch-coverage test suites, `Pythia-rand`, the precision of `Pythia` on the random test suites, `DejaVu-cov`, the precision of `DejaVu` on the branch-coverage test suites, and `DejaVu-rand`, the precision of `DejaVu` on the random test suites. For example, one of the points in `Pythia-cov` is the mean precision of `Pythia` for program `printtok1`, version1, with test suites generated using the coverage technique. These estimates of the means are very accurate, since we had 500 test suites for each.

We performed the following four hypotheses tests:

- H1:** `Pythia-cov` and `DejaVu-cov` have different precisions
- H2:** `Pythia-rand` and `DejaVu-rand` have different precisions
- H3:** `Pythia-rand` and `Pythia-cov` have different precisions
- H4:** `DejaVu-rand` and `DejaVu-cov` have different precisions

In each case the null hypothesis was that the corresponding groups were equally precise.

The data in these groups were not normally distributed so we used non-parametric tests. Since the four variables were all measured for the same 170 program versions, we used the Wilcoxon test [10] for paired variables.

The mean precisions of the four groups were: `DejaVu-rand`: 0.55, `Pythia-rand`: 0.55, `DejaVu-cov`: 0.53, `Pythia-cov`: 0.52. The data did *not* support rejection of the null hypothesis in any of the four tests. (P-values for the tests of hypotheses H1, H2, H3, and H4 were 0.428 and 0.795, 0.410, and 0.434, respectively. Values under 0.05 would have been needed to reject the corresponding null hypotheses.) Thus we conclude that `Pythia` was not significantly less (or more) precise than `DejaVu` and that the nature of the original test suites did not effect the precision of either technique.

4 Further Discussion

Our results and analysis focus on the relative precisions of `DejaVu` and `Pythia`. There are, however, other dimensions of interest when comparing RTS techniques; one involves efficiency.

Our implementations of `DejaVu` and `Pythia` are prototypes, and are not optimized for speed. Further, the implementations had been created independently with no controls imposed to ensure that tasks performed in common were performed by equivalent code, and that only the code associated with differences in the specific algorithms differed. Thus, formal consideration of the relative efficiencies of the techniques is not possible. However, it is still interesting to consider, informally, the execution costs of the two implementations; thus, we measured those costs on the Siemens programs.¹

Table 2 reports our measurements for various aspects of the regression testing and RTS tasks using `Pythia`, `DejaVu`, and the Retest-all technique on the Siemens programs. The table presents results relative to each of the subject programs. For each program, the first six columns of the table show the program name, the time required for `DejaVu` to perform its test-selection analysis (column 1), the time required to execute the test cases selected by `DejaVu` (column 2), the time required for `Pythia` to perform its test-selection analysis (column 3), the time required to execute the test cases selected by `Pythia` (column 4), the time required by the Retest-all technique to select all test cases (column 5). The second-to-last column sums columns 2 and 3, presenting, thus, the overall cost of regression testing using `DejaVu`, and the last column sums columns 4 and 5, presenting, thus, the overall cost of regression testing using `Pythia`. All times are averages, in seconds, of the times observed across all runs on all versions of that program.²

As the data illustrates, the total testing time using `Pythia` was less than the regression testing time using `DejaVu` on six out of seven programs. Measurements on larger programs, with more careful control of non-essential implementation details, are needed to determine how significant the efficiency differences are in

¹We also measured the costs on `Space`, however, due to limitations in our RTS tools, to process `Space` it was necessary to run the two tools on different architectures of quite different speeds; thus, comparisons of the tools' times gathered in these runs on `Space` would be misleading and we do not report them.

²Both `Pythia` and `DejaVu` rely on trace information collected for the existing test suite. As detailed in Section 2, this information must be collected during the *preliminary period* of regression testing – the time prior to when changes are complete and testing has begun. Thus, the costs of instrumentation are not a factor in comparing the critical-phase costs of the techniques, and we do not consider them.

<i>Program Name</i>	<i>DejaVu Analysis Cost</i>	<i>DejaVu Test Execution Time</i>	<i>Pythia Analysis Cost</i>	<i>Pythia Test Execution Time</i>	<i>Retest-all Time</i>	<i>DejaVu Total (Cols. 2+3)</i>	<i>Pythia Total (Cols. 4+5)</i>
totinfo	1.6	0.3	0.5	0.2	0.5	1.9	0.7
schedule1	1.1	0.3	0.8	0.3	0.5	1.4	1.1
schedule2	1.1	0.4	0.8	0.5	0.5	1.5	1.3
tcas	0.9	0.2	0.4	0.3	0.3	1.1	0.7
printtok1	1.5	0.6	1.8	0.6	1.1	2.1	2.4
printtok2	1.4	0.2	1.0	0.2	0.8	1.6	1.2
replace	1.5	0.5	1.0	0.4	1.2	2.0	1.4

Table 2. Average time (seconds) required for various testing and RTS tasks using Pythia, DejaVu, and a Retest-all approach, for branch-coverage-adequate test suites.

practice and to better understand the trade-offs between precision and analysis time. At most, we can say that in the cases studied, the two techniques were observed to be relatively comparable in terms of efficiency.

Another factor illustrated by the data is that, in comparison to the retest-all technique, on the Siemens programs, neither Pythia nor DejaVu saved testing time with respect to the retest-all technique. However, if the test cases used on the Siemens programs were more expensive to execute and validate, then given their precisions, both techniques could have provided savings. Previous studies of DejaVu on programs with more expensive test suites [3, 20] illustrate the possibilities for such savings, and show that they can be practically significant on larger systems.

5 Conclusion

In this paper, we have described the results of a comparative empirical study of two implementations of safe RTS techniques, Pythia and DejaVu. In particular, we compared these tools for precision, measuring their relative success at reducing the number of test cases necessary for regression testing. Our results showed that, despite the differences in their approaches, and despite the theoretically greater ability of DejaVu to select smaller test suites, the two techniques usually selected equivalent test suites.

Of course, if test cases are very expensive to execute, then even the possibility of saving a few more test cases may be worthwhile. For example, one of our industrial collaborators has, for one of its products, a test suite that requires seven weeks to execute, at an average cost per test case of over \$3,000. Such high costs are not ubiquitous; however, when one considers the fully

loaded salaries of engineers in industrial software testing groups, it is easy to see how such costs can be incurred.

On the other hand, when test cases are not particularly expensive, then if DejaVu and Pythia are comparable in terms of analysis cost (a comparability supported by our informal measurements in this study) factors other than relative precision might be more important in selecting which technique to use. One such factor involves relative ease of implementation and availability of supporting tools. Pythia is constructed from commonly available Unix tools, whereas DejaVu requires control flow graphs and profilers that track test execution in change of edges in those graphs. As such, Pythia can be simpler to implement than DejaVu (though this conclusion may also depend on whether existing prerequisite tools are available). A second factor involves the generality of the techniques. As outlined in Section 2, DejaVu operates on abstract program representations, and thus can (and has been) relatively easily extended to handle other languages and operate on other systems. Pythia relies on supporting tools that are C-specific and available on Unix systems; extending it to other languages and systems would be less straightforward.

Of course, these results were obtained on a set of specific programs containing specific 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. In particular, programs and versions containing larger and more complex modifications could produce different results. By coupling such controlled studies with case studies on mature software products, we hope to provide a solid base of data with which to support informed decisions by practitioners.

Acknowledgements

This work was partially supported by NSF Awards CCR-9703108 and CCR-9707792 to Oregon State University and CCR-9870270 to Polytechnic University. Deng Yuetang assisted with the statistical analysis. Tom Ostrand provided the Siemens programs and Alberto Pasquini provided the Space program. We thank the anonymous reviewers for their helpful comments.

References

- [1] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. In *Proc. 3rd Symp. Softw. Testing, Analysis, and Verification*, pages 210–218, December 1989.
- [2] T. Ball. On the limit of control flow analysis for regression test selection. In *Proc. Int'l. Symp. on Softw. Testing and Analysis*, pages 134–142, March 1998.
- [3] J. Bible, G. Rothermel, and D. Rosenblum. Coarse- and fine-grained safe regression test selection. *ACM Trans. Softw. Eng. Meth.*, 10(2):149–183, April 2001.
- [4] D. Binkley. Semantics guided regression test cost reduction. *IEEE Trans. Softw. Eng.*, 23(8):498–516, August 1997.
- [5] Y.F. Chen, D.S. Rosenblum, and K.P. Vo. TestTube: A system for selective regression testing. In *Proc. Int'l. Conf. Softw. Eng.*, pages 211–220, May 1994.
- [6] T.L. Graves, M.J. Harrold, J-M Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *Proc. Int'l. Conf. Softw. Eng.*, April 1998.
- [7] M. J. Harrold, J. Jones, T. Li, D. Liang, A. Orso, M. Penning, S. Sinha, S. Spoon, and A. Gujarathi. Regression test selection for Java software. In *Proc. Conf. O.-O. Programming, Systems, Langs., and Apps.*, October 2001.
- [8] M.J. Harrold, D.S. Rosenblum, G. Rothermel, and E.J. Weyuker. Empirical studies of a prediction model for regression test selection. *IEEE Trans. Softw. Eng.*, 27(3):248–263, March 2001.
- [9] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. Int'l. Conf. Softw. Eng.*, pages 191–200, May 1994.
- [10] R. Johnson. *Elementary Statistics*. Duxbury Press, Belmont, CA, sixth edition, 1992.
- [11] J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test application frequency. In *Proc. Int'l. Conf. Softw. Eng.*, pages 126–135, June 2000.
- [12] J. Laski and W. Szermer. Identification of program modifications and its applications in software maintenance. In *Proc. Conf. Softw. Maint.*, pages 282–290, November 1992.
- [13] H.K.N. Leung and L. White. Insights Into Regression Testing. In *Proc. Conf. Softw. Maint.*, pages 60–69, October 1989.
- [14] H.K.N. Leung and L.J. White. A cost model to compare regression test strategies. In *Proc. Conf. Softw. Maint.*, pages 201–208, October 1991.
- [15] D. C. Oppen. Prettyprinting. *ACM Trans. Prog. Lang. Sys.*, 2(4):465–483, October 1980.
- [16] T.J. Ostrand and M.J. Balcer. The category-partition method for specifying and generating functional tests. *Comm. ACM*, 31(6), June 1988.
- [17] D. Rosenblum and E. J. Weyuker. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Trans. Softw. Eng.*, 23(3):146–156, March 1997.
- [18] G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.*, 22(8):529–551, August 1996.
- [19] G. Rothermel and M.J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Meth.*, 6(2):173–210, April 1997.
- [20] G. Rothermel and M.J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Trans. Softw. Eng.*, 24(6):401–419, June 1998.
- [21] G. Rothermel, M.J. Harrold, and J. Dedhia. Regression test selection for C++ programs. *J. Softw. Testing, Verif., and Rel.*, 10(2), June 2000.
- [22] L. F. Rubin. Syntax-directed pretty printing – a first step towards a syntax-directed editor. *IEEE Trans. Softw. Eng.*, SE-9(2):119–127, March 1983.
- [23] F. I. Vokolos and P. G. Frankl. Pythia: a regression test selection tool based on textual differencing. In *Proc. 3rd Int'l. Conf. on Rel., Quality & Safety of Softw.-Intensive Sys.*, May 1997.
- [24] F. I. Vokolos and P. G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Proc. Int'l. Conf. Softw. Maint.*, pages 44–53, November 1998.
- [25] F.I. Vokolos. *A regression test selection technique based on textual differencing*. Ph.D. dissertation, Polytechnic University, January 1998.
- [26] L. White, V. Narayanswamy, T. Friedman, M. Kirschenbaum, P. Piwowarski, and M. Oha. Test Manager: a regression testing tool. In *Proc. Conf. Softw. Maint.*, pages 338–347, September 1993.
- [27] L.J. White and H.K.N. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *Proc. Conf. Softw. Maint.*, pages 262–270, November 1992.