AN ABSTRACT OF THE THESIS OF

Kent Sayre for the degree of Master of Science in Computer Science presented on August 5th, 1999.

Title: Regression Testing Experiments

Abstract approved: Redacted for privacy

Gregg Rothermel

Software maintenance is an expensive part of the software lifecycle: estimates put its cost at up to two-thirds of the entire cost of software. Regression testing, which tests software after it has been modified to help assess and increase its reliability, is responsible for a large part of this cost. Thus, making regression testing more efficient and effective is worthwhile.

This thesis performs two experiments with regression testing techniques. The first experiment involves two regression test selection techniques, `Dejavu` and `Pythia`. These techniques select a subset of tests from the original test suite to be rerun instead of the entire original test suite in an attempt to save valuable testing time. The experiment investigates the cost and benefit tradeoffs between these techniques. The data indicate that `Dejavu` can occasionally select smaller test suites than `Pythia` while `Pythia` often is more efficient at figuring out which test cases to select than `Dejavu`.

The second experiment involves the investigation of program spectra as a tool to enhance regression testing. Program spectra characterize a program's behavior. The experiment investigates the applicability of program spectra to the detection of faults in modified software. The data indicate that certain types of spectra identify faults on a consistent basis. The data also reveal cost-benefit tradeoffs among spectra types.

Regression Testing Experiments

by

Kent Sayre

A Thesis

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed August 5th, 1999
Commencement June 2000

Master of Science thesis of Kent Sayre presented on August 5th, 1999

APPROVED:

Redacted for privacy

Major Professor, representing Computer Science

Redacted for privacy

Chair of the Department of Computer Science

Redacted for privacy

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for privacy

Kent Sayre, Author

ACKNOWLEDGMENTS

I would like to thank Chengyun Chu for providing the Space test cases for my research, Liu Yi for providing several tools used in my spectra experiments, and the Aristotle research group for providing Siemens program materials.

I would like to thank my major professor, Dr. Gregg Rothermel, for his advice and encouragement throughout my time here at Oregon State University. Also, I'd like to thank my committee members, Dr. Burnett, Dr. Quinn, and Dr. Trehu for their involvement.

Finally, I'd like to thank my parents, Steve and Betsy Sayre, for their support. I thank my girlfriend, Kendra McNair, for enhancing my life tremendously. And I thank Richard Bandler for showing me a completely new direction in life.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

REGRESSION TESTING EXPERIMENTS

Chapter 1

INTRODUCTION

Software plays an integral role in our everyday lives. With our increasing reliance on software, it is important that software behave according to its specifications. A software malfunction may range from a nuisance (e.g., your word processing program crashes while you are editing a trivial document) to a catastrophe (e.g., software responsible for navigating an airplane fails and causes an accident). Therefore, the reliability of software is essential, and perhaps the most important tool in helping increase and assess such reliability is software testing [4].

After software is released, it will almost inevitably be modified. Such modification is referred to as software maintenance. While software maintenance is obviously important, it also is expensive to perform. The budget allocated to software maintenance can be up to 80% of the cost of the software throughout its lifecycle according to some estimates [15]. Other studies place the cost of software maintenance at up to two-thirds of the overall cost of software [16, 28].

All this maintenance requires retesting of the software and we call that testing *regression testing.* Regression testing is testing that is performed after modifications have been made, to help assess and increase the reliability of a program. If the program exhibits failures under regression testing, the program is debugged to locate and correct the faults responsible for these failures. Then the program undergoes regression testing again. This regression testing cycle continues until the program has an adequate degree of reliability. The regression testing strategy has two parts: first, it must verify that functionality that was meant to be unchanged remains unchanged, and second, it must assess whether faults have been introduced into modified areas of the program [35]. Regression testing may account for up to 50% of software maintenance cost [4, 13]. Investigating techniques for improving the effectiveness and efficiency of regression testing is the primary goal of this thesis.

Regression testing is different from other testing in that at regression test time, a test suite has already been constructed for the program. One way to regression test would be to simply rerun the entire test suite created for the original program on the modified program. Rerunning all tests on the modified program, however, may actually not be necessary. Instead, we can intelligently select only the subset of tests that traverse modified sections of code [26]. This test selection method will only be valuable if the combined costs of creating the new subset of tests for retesting and running the newly created subset of tests is less than the cost of simply rerunning the entire test suite.

A second area of research with potential impact for regression testing involves program spectra. A program spectra is a signature of a program's dynamic behavior in which the frequency of execution of each program component (e.g., statement, branch, path) is registered. Program spectra rely on path profiling to characterize a program's behavior [17]. These spectra can be used to characterize a program's execution on a set of tests. Regression testing may potentially be made more effective and efficient through the use of program spectra. Where a modified program's spectra differ from the original program's spectra, this may be an indication of the presence of a fault. Furthermore, the spectra may point to the areas in the modified code most likely to contain the fault. Essentially, program spectra might help software engineers find and fix faults in modified program versions more quickly than they can presently because spectra guide them to the faults.

A study done in 1998 [36] examined 612 papers out of all issues of three software engineering journals published in 1985, 1990, and 1995. The researchers found that one-third of these papers lacked any sort of empirical verification, one-third provided only informal verification, and fewer than 10 percent offered formal, rigorous empirical validation. Based on data such as this, it has been argued that computer science must become like other scientific disciplines regarding its experimental approach in order to actively keep developing as a discipline [29]. Too often in computer science, claims are made without being empirically verified. Providing empirical validation of theories in a paper can greatly enhance the results and support the plausibility of the ideas.

Understandably, there are difficulties in experimenting. Researchers may be overwhelmed at the task of developing infrastructure that will support different experiments. Representative software subjects may be impossible to find. Nevertheless, computer scientists should experiment.

Therefore, this thesis performs two experiments to further our knowledge about the efficiency and effectiveness of regression testing techniques presented in the literature. The first portion of this thesis concentrates on two specific regression test selection techniques: `Dejavu` and `Pythia`. Regression test selection techniques attempt to select a subset of tests from the original test suite that expose differences in output between the orginal program and a modified version. `Dejavu` is a test selection technique that uses control flow graphs as the basis for such selection [24]. In contrast, `Pythia` is a test selection technique that uses textual differencing for such selection [31]. The data show that `Dejavu` can occasionally select smaller test suites than `Pythia`. With respect to efficiency, however, `Pythia` usually outperformed `Dejavu`.

The second portion of this thesis involves the investigation of program spectra as a tool to enhance regression testing. It investigates a question fundamental to the applicability of program spectra to detection of faults in modified software. We found that certain spectra types exhibit spectral differences often when there is a fault executed on a program. The data also reveal cost-benefit tradeoffs among the various spectra types.

The rest of this paper proceeds as follows. Chapter 2 provides the requisite background for understanding the rest of the paper. Chapter 3 presents the first empirical study: the comparison between two separate test selection techniques. Chapter 4 presents the second empirical study: the analysis of using program spectra on a large scale, industrial program. Chapter 5 concludes the paper giving a summary and suggestions for future work.

Chapter 2

BACKGROUND

This chapter provides background information necessary to understand the rest of the paper.

2.1 Control Flow Graphs

A *control flow graph* (CFG) is a directed graph in which each node represents a statement or a *basic block* (single-entry, single-exit sequence of statements) in a procedure [1]. The directed edges between nodes depict flow of control in the program. A predicate node is a node where control flow has a choice of outgoing edges based on the state of the program at the time. A control flow graph has both entry and exit nodes, with an entry node being the first node in the graph representing entry into the procedure and an exit node representing control leaving the procedure. As an example, Figure 2.1 presents the CFG for a program `Sums` [11].

2.2 Code Instrumentation

Code instrumentation is the process of inserting probes into code to discover what sections of code are executed by a given test case. A primitive method of code instrumentation is to insert print statements that write, to standard

```
program Sums
1 read i
2 sum = 0
3 while i < 10
4   read j
5   sum = sum + j
6   i = i + 1
  endwhile
7 print sum
end Sums
```

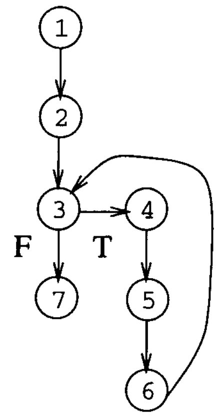

FIGURE 2.1: Program **Sums** and its control flow graph.

output, which statements, basic blocks, functions, and so forth are executed. A more sophisticated method, the method we employ, is to have the instrumented program record to a file which parts of code were executed by a test case. A *branch trace* is a record of the branches in a CFG traversed by an instrumented program. An *edge trace* is a record of the edges in a CFG traversed by an instrumented program. Tracing is part of a larger idea entitled program profiling. When profiling a program, we can track the behavior of it in a controlled manner as it executes on a particular test case. Reference [3] discusses program profiling more thoroughly.

2.3 Regression Testing

Regression testing is the testing of software after modifications have been made, and is performed in order to assess and increase the reliability of that software.

Let P denote a program, P' a modified program and T the original test suite created to test P. A typical regression testing process is as follows [26]:

1. Select $T' \subseteq T$, a set of tests to execute on P'.

2. Test P' with T', establishing P''s correctness with respect to T'.

3. If necessary, create T'', a set of new functional or structural tests for P'.

4. Test P' with T'', establishing P''s correctness with respect to T''.

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

Step 1 involves the *regression test selection problem*: the problem of selecting a subset of T' of T with which to test P'. Step 3 involves the *coverage identification problem*: the problem of determining what sections of P' need additional testing. Steps 2 and 4 address the *test suite execution problem*: the problem of efficiently executing tests and checking test outputs for correctness. Step 5 addresses the *test suite maintenance problem*: the problem of updating and storing test information. Although each of these activities is important, in this paper, we concern ourselves with Steps 1, 2, and 4.

Reference [24] describes two phases of regression testing: a *preliminary* phase and a *critical* phase. The preliminary phase is the phase in which software engineers are modifying the software before the new version is released. The critical phase comes after the preliminary phase, and is the phase after modifications are complete, in which testing must be performed so the software can be re-released. Intelligent regression testing strategies attempt to schedule as many regression testing activities as possible in the preliminary phase. Simply put, this lessens the burden of testing during the critical phase. Doing this leads to less time spent in the critical phase and helps prevent testing-related delays in distribution of the software.

There are two ways to fit a two-phase process into an overall regression testing strategy. A *big bang* approach performs all the modifications and then follows with regression testing. An *incremental* approach regression tests the software as much as possible incrementally, whenever a change or group of related changes is made [24].

While regression test selection techniques may vary widely in how they function, they have the same goal: to select an adequate subset of tests from the original test suite to be rerun on the modified program. To rationally compare such techniques, [22] presented a framework, consisting of the following four categories: *inclusiveness*, *precision*, *efficiency*, and *generality*.

Inclusiveness is defined as the degree to which the regression test selection technique chooses tests that reveal different behavior in the modified software. A 100% inclusive technique is called a *safe* technique.

Precision is defined as the degree to which the regression test selection technique omits tests that do not reveal different behavior in the modified software. In theory, we would like techniques to be 100% precise, but this is impossible because the problem of identifying exactly these tests is undecidable. The retest all strategy can be thought of as 100% imprecise or 0% precise because it essentially chooses all of the tests and reruns all of them, omitting none of the tests that do not reveal different behavior.

Efficiency is defined as the measure of the temporal requirements needed for the technique and includes the time required for analysis and the time required to execute selected test cases.

Generality is defined as the degree to which the regression test selection technique can be applied to a broad range of programs, programming languages, operating environments, and so forth.

Chapter 3

EMPIRICAL STUDY 1: RTS TECHNIQUES

3.1 Introduction

As discussed in Section 2.3, regression test selection (RTS) techniques are techniques that select a subset of an original test suite for rerunning on a modified program. These techniques can be valuable if the cost of performing the regression test selection analysis and rerunning the selected subset of test cases on the modified program is less than the cost of simply rerunning the whole original test suite on the modified program.

Our attention in this study focuses on two *safe* regression test selection techniques: techniques that select all of the test cases, from the original test suite, that may reveal different output behavior[1]. To locate these existing test cases, safe techniques select all of the test cases, from the original test suite, that may traverse modified sections of code in the modified program. (These techniques really seek to select only the test cases that will reveal different output behavior, but that is impossible, so they select a superset of test cases, all the modification-traversing test cases, to guarantee they have selected all the test cases that may reveal different output behavior.) They may also inadvertantly

[1] These techniques can only be *safe* if certain conditions are met. The parameters of the regression test must remain constant and only the program versions may differ in the controlled regression testing strategy necessary for these techniques to work properly.

select some test cases that do not traverse modified sections of code, yet they attempt to avoid doing this because rerunning and validating test cases is costly.

We investigated two safe techniques in particular: `Dejavu` and `Pythia`. The `Dejavu` technique uses control flow graphs for its analysis and selection of test cases from the original test suite [24]. `Dejavu` provides one of the most precise safe techniques currently available. It has performed well regarding efficiency and precision in empirical studies reported in [5, 18, 23, 24, 25, 26].

`Pythia` is an RTS technique, different from `Dejavu`, that uses textual differencing for its test selection method [31, 32]. In [32], it is claimed that `Pythia` is safe and nearly as precise as `Dejavu`. Also it is said that `Pythia` may be more efficient than `Dejavu`.

There are many cost-benefit tradeoffs involved in using an RTS technique. For example, if a technique spends a lot of time on analysis, it may be relatively more precise but relatively less efficient than another technique. The converse is also true: a technique that spends relatively little time on analysis may be relatively more efficient and relatively less precise than another technique. An RTS technique that lacks generality may be very precise and safe for a particular type of software. (Refer to Section 2.3 for terminology definitions.)

This study explores these tradeoffs and the claims made in [32] through an empirical study of `Pythia` and `Dejavu`. The chapter proceeds as follows. It begins with a description of `Dejavu` and `Pythia`. Next, it describes in-depth the experiment design, and presents data and analysis. Finally, it presents conclusions.

3.2 Background

3.2.1 Dejavu

`Dejavu` first builds control flow graphs (CFGs) for a given program P and a modified program P'. It is assumed that the existing test suite T has previously been executed on an instrumented version of P, and a record was made of which test cases in T traverse which particular edges in the CFG for P. A depth-first search follows simultaneously of both control flow graphs, comparing program statements associated with CFG nodes of P and P'. If a pair of nodes N and N' in the CFGs have associated code that is not lexicographically equivalent, the algorithm selects all test cases from T that, in P, reached N [18, 21]. `Dejavu` runs under UNIX and currently executes on C programs. Using `Dejavu` requires `Aristotle`, a system for research on and development of program analysis based tools [9, 10].

3.2.2 Pythia

Unlike `Dejavu`, which uses a control flow graph, `Pythia` uses *textual differencing.* Textual differencing works by comparing the program source file text directly, without utilizing an abstract representation of the program. This method uses the compiler to instrument the base program and create history files of which specific test cases traversed which basic blocks of the base program. It then performs a comparison between the base program and the modified version and selects all the test cases that traverse modified basic blocks in the modified version. Frankl and Vokolos' goal in designing this tool was to create a tool that was safe and balanced precision, efficiency and the ability to support large, industrial scale software systems. The tool functions under UNIX and executes on C programs [30, 31, 32]. `Pythia` requires only standard UNIX utilities for operation and a compiler that can perform instrumentation.

In selecting test cases, `Pythia` will *always* select a superset of test cases compared to the set of test cases `Dejavu` selects. The reason behind this is that `Dejavu` functions on the statement level so it is provably more precise in selecting test cases whereas `Pythia` functions on the level of basic blocks. `Dejavu`'s finer granularity provides for more precision and guarantees that `Pythia` will select a number of test cases greater than or equal to the number of test cases `Dejavu` selects.

3.2.3 Related Work

There have been many different regression test selection techniques proposed. Of those techniques proposed, however, few have been empirically studied. Some studies [7, 19, 20, 24, 25, 34] have previously shown that RTS techniques can be valuable, yet their costs and benefits can not be adequately compared through these studies because each study used different programs, program versions, and test suites. To accurately assess the relative costs and benefits of the RTS techniques, it is necessary to hold other factors constant while varying only the techniques. There have been only two comparative empirical studies reported in the literature [8, 18]. Of the two, only one [18] compared safe techniques, but that study considered only relative precision. This study compares two safe regression test selection techniques for both precision *and* efficiency.

3.3 The Experiment

3.3.1 Objectives

We are interested in the following research questions:

1. How do `Dejavu` and `Pythia` compare to one another in terms of precision: is either tool more adept than the other at selecting smaller test suites?
2. How do `Dejavu` and `Pythia` compare to one another in terms of efficiency?

3.3.2 Measures

To address our first research question, we will measure the percentage of tests selected by `Dejavu` and `Pythia` over a variety of programs and test suites. Then, we will compare these numbers and base our answer to this question on that analysis.

To address our second research question, we will measure the efficiency of `Dejavu` and the efficiency of `Pythia` over a variety of programs and test suites. For each tool, on each given program, modified version, and test suite, we will record the time required to execute the tool, and the time required to run the set of tests selected by the tool, and add these times. We will then compare these times calculated for the two techniques and base our answer to this question on that analysis.

3.3.3 Subjects

We used eight C programs in our experiment, including seven programs collectively known as the "Siemens programs" and one program known as *Space.* They are each described below.

3.3.3.1 Siemens Subjects

The Siemens programs are known as such because they were initially assembled for and used by Siemens Corporate Research in a study of dataflow and controlflow-based test adequacy criteria [12]. For each subject base program, the researchers at Siemens created a large *test pool* full of possible test cases for the program. First, they created a set of black-box test cases, using the *category partition method* and the Siemens Test Specification Language tool [2, 14]. After creating these black-box test cases, the researchers manually created white-box test cases to ensure coverage of each executable statement, edge, and definition-use pair in the program or its control flow graph by at least thirty test cases.

The researchers sought to introduce faults into the subject programs that were as realistic as possible. Most seeded faults involve single line changes while a few involve multiple lines. The researchers excluded faults that were not detected by at least three test cases in the test pool and no more than 350 test cases. Table 3.1 includes information on the programs, including numbers of functions, lines of code, numbers of versions, test pool size, average test suite size and descriptions of functionality.

To support our experimentation, we used the Siemens test pools to generate two different types of test suites: test suites that were *edge-coverage-adequate*[2] and test suites that were randomly selected. The edge-coverage-adequate test suite pool contained 1000 test suites. Each test suite within the edge-coverage-adequate test pool covered all the edges of the CFG of the program. The

---

[2] Edge-coverage-adequacy implies that all edges in a control flow graph of a program are traversed by a test suite.

Program Name	*Number of Functions*	*Lines of Code*	*Number of Versions*	*Test Pool Size*	*Average Test Suite Size*	*Description of Program*
totinfo	7	346	23	1052	7.2	information measure
schedule1	18	299	9	2650	8.3	priority scheduler
schedule2	16	297	10	2710	7.8	priority scheduler
tcas	9	138	41	1608	5.7	altitude separation
printtok1	18	402	7	4130	16.3	lexical analyzer
printtok2	19	483	10	4115	11.8	lexical analyzer
replace	21	516	31	5542	18.8	pattern replacement
space	135	9126	38	13585	155	parses antenna-array description language

TABLE 3.1: Subject programs.

random-coverage test suite pool also contained 1000 test suites. The random-coverage test suites each had the same size as their counterparts in the edge-coverage-adequate test suite pool but each random-coverage suite was generated by randomly selecting the same number of test cases from the test universe as the corresponding edge-coverage-adequate test suite. Average test suite sizes for the programs are reported in Table 3.1.

The Siemens programs and their respective test suites have several advantages. They were relatively easy to obtain because the Siemens group had made the programs and test cases available to fellow researchers. Due to the manner of their construction, the seeded faults within the programs do model real world faults. By using subjects from an external source, we reduce the potential for bias. The subjects have also been used previously in other studies [12, 27].

3.3.3.2 Space Program

The *Space* program has 9,126 lines of C code (including comments), and was developed for the European Space Agency. The purpose of the *Space* program is to "provide a language-oriented user interface that allows the user to describe the configuration of an array of antennas using a high level language" [6]. An Array Definition Language was created and used within the program. It enables the user to describe a particular antenna array through fewer statements instead of writing the complete list of elements, positions, and excitations [35]. Three subsystems comprise the *Space* program: *parser*, *computation*, and *formatting*. Elaboration on the subsystems can be found in [6].

Space came to us with a test pool containing 10,000 test cases; these test cases had been randomly generated for use in a previous study [32]. Unfortunately, these test cases failed to cover all the code. Therefore, new test cases were created until each reachable node and edge in the CFG of the program was covered by at least 30 test cases. After this addition, the final test pool contained 13,585 test cases. A greedy algorithm was then used to build 1000 edge-coverage-adequate test suites. The algorithm would select a test case and if it added coverage, add it to the suite. Otherwise, it was discarded. The algorithm executed until the test suite contained test cases that covered all edges of the CFG of the program. The 1000 random coverage test suite pool was created by selecting randomly a fixed number of test cases from the entire selection of inputs corresponding to each edge-coverage-adequate test suite. The average size of the test suites created by this method was 155.

Finally, we randomly sampled 500 of the test suites of each kind (edge-coverage-adequate, random-coverage) to create smaller test suite pools. We did this by randomly generating a number, then selecting that suite from both the larger test pools and putting those respective test suites into the smaller test pools. We followed the same method in creating these test suites for *Space* as we did with the Siemens subject programs.

An advantage of *Space* as a subject is that it was provided with 33 faulty versions, discovered by the developers of the program during its creation. Our new tests uncovered 5 additional faulty versions, giving us a total of 38 faulty versions. However, we ultimately experimented with 34 faulty versions due to problems with the `Pythia` tool.

3.3.4 Experiment Design

The experiment procedure is presented in Figure 3.1.

Experiment Procedure RTS experiment procedure
Input: Test Selection Techniques: `Dejavu`, `Pythia`
Test Pools: *Edge-coverage-adequate, Random-coverage*
Subject Programs: *Print Tokens, Print Tokens2, Schedule, Schedule2, Replace, Tcas, Tot Info, Space*
Output: Raw experiment results

```
begin
    for each technique ∈ Test Selection Techniques
        for each subject ∈ Subject Programs
            for each suite ∈ Test Pools
                Run technique on subject with suite
            endfor
        endfor
    endfor
end
```

FIGURE 3.1: RTS experiment procedure

The experiment involved eight programs with two test pools (an edge-coverage-based pool and a random-coverage pool), and two different test selection techniques, `Dejavu` and `Pythia`.

We applied both RTS techniques to each subject program with each of the test suites in the two test pools[3].

The independent variables in this experiment are: the various subject programs, the test suites, and the two regression test selection techniques.

The dependent variables in this experiment are: the size of the selected test suite, the running time (analysis time and time to select test cases) of the test selection tools, and the time required to run selected test cases. We measure each of these variables for each test suite, regression test selection technique, and subject program.

3.3.4.1 Experiment Instrumentation

This experiment has the advantage of using the actual implementations of `Pythia` and `Dejavu` created by the researchers who developed the techniques. Rothermel provided an implementation of `Dejavu` for our use in this experiment. Because `Dejavu` was already configured to execute in our operating environment, it did not require any modification to execute. Vokolos provided an implementation of `Pythia` for our use. It required only a few system-dependent modifications (e.g., path name to compiler) to make it functional.

---

[3] In the case of *Space*, only edge-coverage-adequate suites were used, due to time constraints.

To verify that these tools worked properly, we ran several smaller experiments on each tool and verified that the results achieved were as expected.

3.3.4.2 Experiment Method

Below, we describe the methods used to run `Dejavu` and `Pythia` in our study.

To run `Dejavu`, we:

1. Used `Aristotle` to construct the control flow graph for P.

2. Used `Arisotle` to instrument P.

3. Ran all test cases 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 of P_i.

(b) Ran `Dejavu` on P, P_i, and H.

(c) Ran and validated outputs for all test cases in T on P_i.

(d) Ran and validated outputs for all selected test cases on P_i.

To run `Pythia`, we:

1. Used `Pretty` (a program formatter) to translate the source files for the old version of the program into canonical form.

2. Instrumented and compiled the canonical files, i.e., the source files in canonical form.

3. Ran all test cases in T on P and obtained their basic block execution traces.

4. Used `Pretty` to translate the modified source files (i.e., source files for all the P's) into canonical form.

5. Used `Pythia` to analyze the differences between the old and new canonical files for P and each P' and select all the tests that exercised basic blocks that had been modified in P'.

We automated the execution of the experiments via UNIX shell scripts. Both the precision and efficiency information of the two RTS techniques were output by the script. After running the techniques in this manner, we used an analysis tool to transform our raw data files into a tabular summary of the results.

To perform the `Pythia` experiment, we first had to use a program formatting tool on the source code (steps 1 and 4). Ideally, the source code would automatically go through the `Pythia` tool and be transformed into canonical form. Unfortunately, some C code constructs could not be handled by the pretty-printer so we had to resort to manually removing the constructs, running the source through the tool, then adding the constructs back. Thus, we performed this step as a preprocess on all source files, rather than in the automated scripts. `Dejavu` does not require any such pretty-printer tool. However, to maintain consistency among the experiment, we performed both experiments with "prettied" subject programs.

3.3.5 Threats to Validity

There are several threats to validity for this experiment that must be taken into account when assessing its results. There are *external* threats to validity:

factors that limit the ability to generalize the results of the study to a larger set of software subjects. One external threat is that the subject programs—the Siemens programs and the *Space* program—are not necessarily representative of a general class of programs[4]. Similarly, the faults within the subject programs may not be representative of the types of faults that occur in most programs. To reduce these threats, this study must be repeated with different subjects.

There are also *internal* threats to validity: influences that can affect the dependent variables without the researchers' knowledge. The main internal threat to validity is the threat of instrumentation effects. Stated another way, the software tools used to instrument the subject programs could have unknowingly changed the actual way the programs execute so that they behave differently, and consequently bias the results of the study. To minimize internal threats to validity, we performed several validity checks on our results, including examining results for conformance, and examining them to ensure that `Pythia` selected more tests than `Dejavu`. No efforts were made, however, to control for the structure of the source programs or for the area where changes in the programs happened.

3.4 Data and Analysis

The strategy used to analyze the data generated from the experiment is to calculate the efficiency and precision of each test selection tool (`Dejavu` and `Pythia`). After calculating these, we will compare the data to assess which

---

[4] Any chosen program would necessarily have this external threat to validity because the software engineering research community has not established a benchmark suite of programs.

Program Name	*Dejavu Analysis Cost*	*Dejavu Test Execution Time*	*Pythia Analysis Cost*	*Pythia Test Execution Time*	*Retest-all Time*	*Dejavu Total (Cols. 2+3)*	*Pythia Total (Cols. 4+5)*
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
space	18.0	9.7	5.6	3.5	28.6	27.7	9.1

TABLE 3.2: `Dejavu` vs. `Pythia` vs. Retest-all: Average time (seconds) comparison between methods for each program over its set of versions for **Edge-coverage-adequate** test suites.

Program Name	*Dejavu Analysis Cost*	*Dejavu Test Execution Time*	*Pythia Analysis Cost*	*Pythia Test Execution Time*	*Retest-all Time*	*Dejavu Total (Cols. 2+3)*	*Pythia Total (Cols. 4+5)*
totinfo	2.0	0.4	0.5	0.2	0.6	2.4	0.7
schedule1	1.3	0.4	0.8	0.4	0.6	1.7	1.2
schedule2	1.3	0.6	0.8	0.6	0.6	1.9	1.4
tcas	1.1	0.3	0.3	0.1	0.5	1.4	0.4
printtok1	1.7	0.7	1.8	0.7	1.4	2.4	2.5
printtok2	1.6	0.3	1.0	0.3	1.0	1.9	1.3
replace	1.7	0.6	1.0	0.5	1.4	2.3	1.5

TABLE 3.3: `Dejavu` vs. `Pythia` vs. Retest-all: Average time (seconds) comparison between methods for each *Siemens* program over its set of versions for **Random-coverage** test suites.

technique is relatively more efficient than the other and which technique is relatively more precise than the other.

Tables 3.2 and 3.3 present efficiency results for `Dejavu`, `Pythia`, and the retest-all technique for edge-coverage-adequate and random-coverage test suites, respectively.

`Pythia`'s efficiency was better than `Dejavu`'s in six of the seven *Siemens* programs for the edge-coverage-adequate test suites. The largest single difference between Pythia and Dejavu was 1.2 seconds while the smallest single difference between the two techniques was .2 seconds. The exception was when Dejavu was more efficient than Pythia in the case of *Print Tokens*, with a difference of .3 seconds. For the subject *Tot Info*, Pythia took less than half the time (.7 seconds) of Dejavu (1.9 seconds). For *Space*, Pythia also outperformed Dejavu in terms of efficiency by a time of 9.1 seconds to 27.7 seconds respectively.

Pythia's efficiency was better than Dejavu's in six of the seven *Siemens* programs for the random-coverage test suites as well. The largest single difference between Pythia and Dejavu was 1.7 seconds while the smallest difference between the two techniques was .1 seconds. For the subject *Tot Info*, Pythia took less than one-third of the time (.7 seconds) of Dejavu (2.4 seconds). The exception was when Dejavu was more efficient than Pythia in the case of *Print Tokens*, with a difference of .1 seconds.

Among the three RTS techniques listed, the retest-all technique performed more efficiently than both Dejavu and Pythia for all seven *Siemens* subject programs for the edge-coverage-adequate test suites. The time difference between retest-all and the other two RTS techniques ranged from .2 seconds to 1.4 seconds. In the case of *Space*, however, both RTS techniques outperformed the retest-all technique for edge-coverage-adequate suites. Dejavu took 27.7 seconds, Pythia 9.1 seconds, and retest-all 28.6 seconds.

Among the three RTS techniques listed, the retest-all technique performed more efficiently than both Dejavu and Pythia for six of the seven *Siemens* programs for the random-coverage test suites. The exception was when Pythia's total execution time (.4 seconds) was less than the retest-all technique's time (.5 seconds) for *Tcas*.

Disregarding this exception, the time difference between retest-all and the other two RTS techniques ranged from .1 seconds to 1.8 seconds.

The relationship of `Dejavu` and `Pythia` to retest-all is worth commenting on. With test suites as small as the *Siemens* test suites, analysis cost is a large percentage of the total cost of running the RTS techniques, so that is why the RTS techniques are apparently not worthwhile at this small scale. If the *Siemens* test suites were larger or if the tests required more time to execute or validate, the RTS techniques would become more cost effective in terms of efficiency than the retest-all technique. The fact that on *Space*, with larger test suites, `Dejavu` and `Pythia` outperform retest-all, supports this suggestion, as do empirical results for `Dejavu` presented elsewhere [8, 26].

Also, we believe that `Dejavu`'s efficiency would eclipse the efficiency of the current implementation of `Pythia` at some test suite size because that implementation is based on interpreted PERL [33] scripts whereas `Dejavu` is a binary program. Implementation of a compiled version of `Pythia` could address this problem, although that would be a departure from `Pythia`'s design philosophy.

With respect to precision for the edge-coverage-adequate test suites, the two RTS techniques almost always selected the same percentage of test cases for the *Siemens* programs. However, for *Space*, the two RTS techniques *always* selected the same percentage of test cases for each of the 34 measured versions. Figure 3.2 shows precision results for these test suites, plotting the average percentage of tests selected by `Dejavu` against the average percentage selected by `Pythia` over the edge-coverage-adequate test suites for all subject programs. Virtually all points lie on the x=y line, indicating equivalent precision.

Altogether, there were only 4 modified versions of the entire 131 modified *Siemens* versions used where `Dejavu` selected a smaller percentage of tests than

`Pythia`. There were two versions of *Print Tokens* where `Pythia` selected 100.0 percent and 87.6 percent of the tests while `Dejavu` selected only 63.8 percent and 86.9 percent respectively. There were two versions of *Tot Info* where, on average over the 500 test suites, `Pythia` selected 92.6 percent and 100.0 percent of the tests while `Dejavu` selected 85.3 percent and 99.5 percent, respectively. None of the 34 *Space* versions had a difference in precision between `Dejavu` and `Pythia`.

With respect to precision for the random test suites, the two RTS techniques also almost always selected the same percentage of tests for the *Siemens* programs[5]. Figure 3.3 shows these results in a manner similar to Figure 3.2. In this case, there were 5 modified versions of the entire 131 modified *Siemens* versions used where `Dejavu` selected a smaller percentage of tests than `Pythia`. There were the same versions of *Print Tokens*, as described above, where `Pythia` selected 100.0 percent and 97.4 percent of the tests while `Dejavu` selected only 53.3 percent and 96.0 percent. There were also the same two versions of *Tot Info* where, on average over the 500 test suites, `Pythia` selected 91.8 percent and 100.0 percent of the tests while `Dejavu` selected 85.2 percent and 99.8 percent respectively. Additionally, `Pythia` selected 8.1 percent of the tests on average for a version of the subject program *Tcas* whereas `Dejavu` selected 8.0 percent.

Obviously, the retest-all technique is excluded in discussing precision because it always selects all the tests.

It is interesting to note the variability in number of tests selected. The average percentage of tests selected for both RTS techniques (`Dejavu` and `Pythia`),

---

[5] Sizes of selected tests were measured only for the *Siemens* programs due to time constraints.

for the edge-coverage-adequate suites ranged from 5.5 percent up to 100 percent. For the random-coverage suites, these averages ranged from 1.0 percent of tests selected up to 100 percent of tests selected.

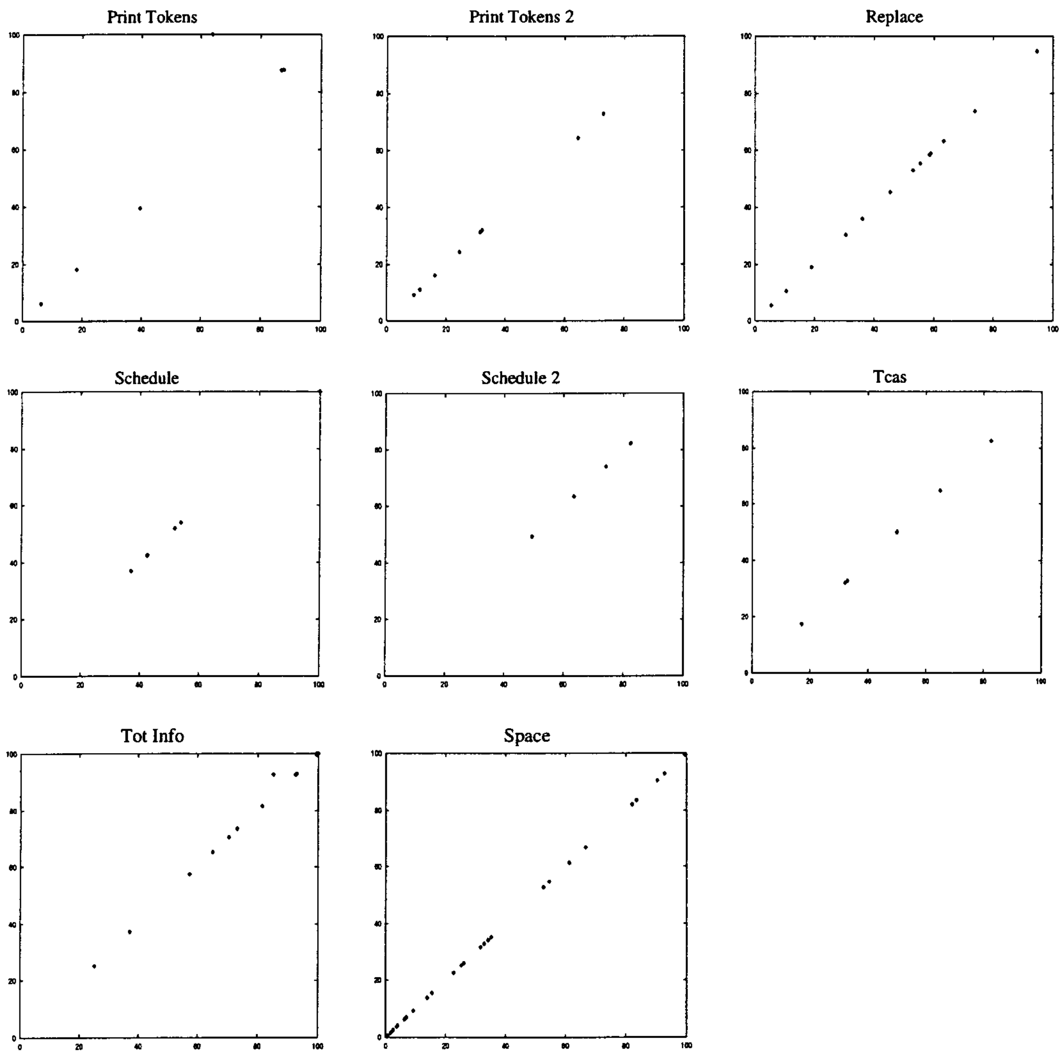

FIGURE 3.2: Precision information graphs for **edge-coverage-adequate** test suites. The horizontal axis represents the percentage of tests selected by `Dejavu`. The vertical axis represents the percentage of tests selected by `Pythia`. Each point represents the average percentage of test cases selected over a particular version.

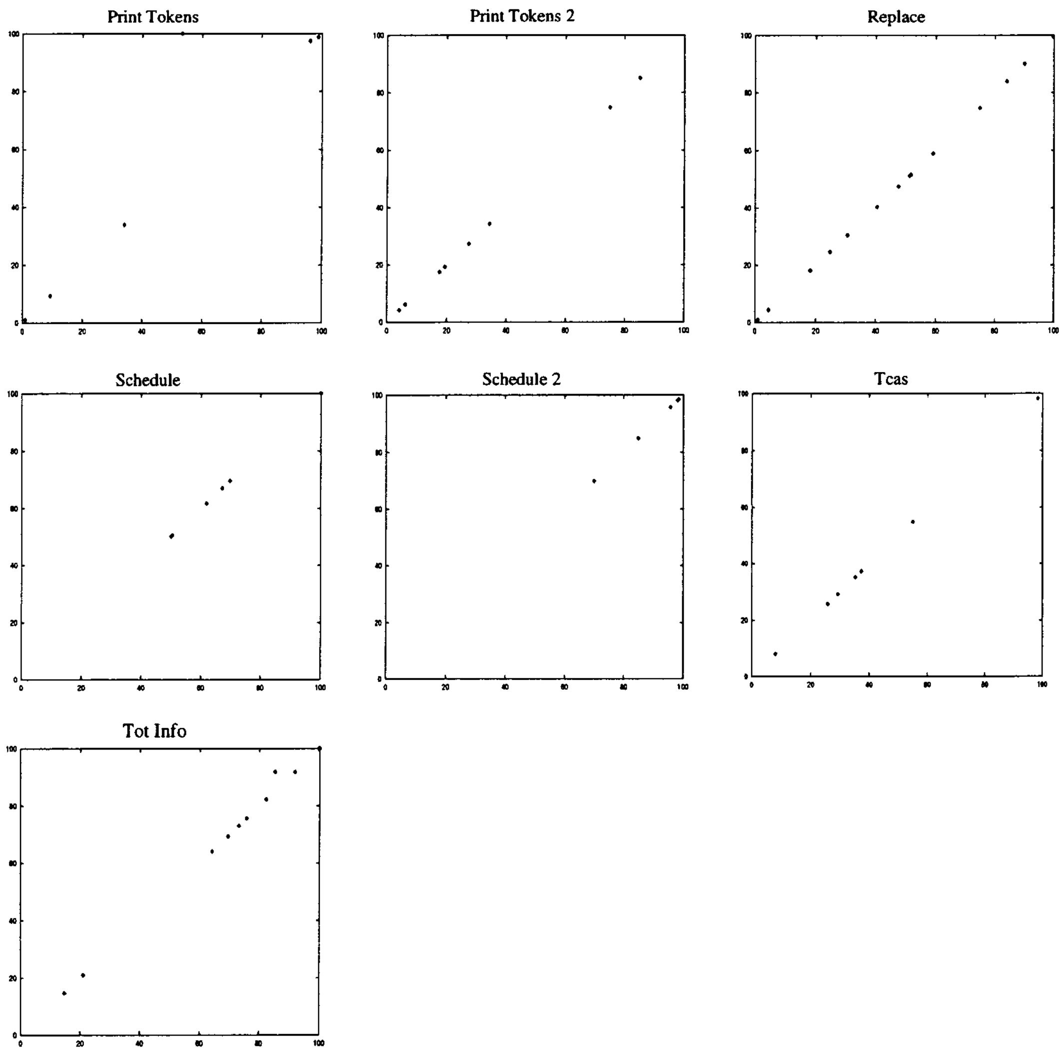

FIGURE 3.3: Precision information graphs for **random-coverage** test suites. The horizontal axis represents the percentage of tests selected by `Dejavu`. The vertical axis represents the percentage of tests selected by `Pythia`. Each point represents the average percentage of test cases selected over a particular version.

3.5 Conclusion

In this study, we investigated two *safe* RTS techniques—`Dejavu` and `Pythia`—and measured their degrees of efficiency and precision. We discovered that, in the cases we examined, `Pythia` generally costs less than `Dejavu` in terms of total time to execute.

We also discovered that, for the cases we examined, `Dejavu` was occasionally, but not often, more precise than `Pythia`. In the few cases in which `Dejavu` was more precise and selected a smaller percentage of tests, the difference in percentage was not substantial.

Both `Dejavu` and `Pythia` suffer in terms of efficiency on the *Siemens* programs, in comparison to retest-all; however this is a result of the small size of the *Siemens* test suites. Had these test suites been of larger size or contained tests that required more time to execute, both techniques would have made gains in efficiency. However, the results on *Space* demonstrate that the two RTS techniques can be more efficient than retest-all.

If these results generalize, the implication for testers doing regression testing is that they must consider their testing situation before deciding on the appropriate RTS technique. If test suites are large or test cases require a lot of time or human effort to execute and validate, it may behoove the tester to use `Dejavu` for its superior precision. Otherwise, `Pythia` probably would be the preferred method.

Chapter 4

EMPIRICAL STUDY 2: INVESTIGATION OF PROGRAM SPECTRA

4.1 Introduction

A *program spectra* is a signature of a program's dynamic behavior in which the frequency of each program component (e.g., statement, branch, path) is registered. Program spectra were first proposed by Reps et al. [17] as a heuristic for understanding differences in program executions. Specifically, Reps et al. propose using path spectra to aid in the identification and correction of year 2000 faults. Given a program, run under different operating conditions (such as different system dates), path spectra for the programs should be different if execution is affected by the different operating conditions; identifying where the spectra differ provides a software engineer with a starting point for locating faults in the program. Use of this heuristic should allow software engineers to find the code that cause faults more easily and quickly than otherwise.

The motivation for program spectra is to aid in the testing and debugging of software by using various path profiling techniques to provide information. After a program has been instrumented by a path profiler, the number of times a program component such as a statement or partial path executes can be recorded for a given run. Each execution of the program results in a path spectrum for the execution of the program. This path spectrum provides the distribution of program components traversed throughout the last execution of the program.

The primary application of spectra presented in [17] involves comparing path spectra from different runs of the same program. If different runs generate different spectra, the spectral differences may be used to identify paths in the program where control diverges between the two runs. By selecting input data to keep all factors constant but one, the divergence in control between the two runs can be attributed to this varied factor. The point of divergence will be where the software engineer looks first in the hunt for the cause of different behavior.

Reference [17] also suggests, without further investigation, an application of spectra to regression testing parts of a system affected by a modification. The suggestion is to compare path spectra to provide information about the extent of changes in behavior of a program. The notion is that doing a path-spectrum comparison may allow the software engineer to realize the actual magnitude of the behavior differences that a modification introduces.

In regression testing, theoretically all factors that affect program execution will be held constant except the difference in the two programs. This means that two different programs, a base version and its modified version, will each run the same test suite and generate program spectra. Any difference in program spectra will necessarily be a result of modifying the program. Reps et al. claim [17] that the presence of such differences will indicate the presence of regression faults, and that spectra differences can help software engineers locate these faults.

For spectra to be useful in regression testing as suggested in [17], the presence of spectral differences must be a good indicator of the presence of faults. Reps et al. [17] do not investigate this. Thus, in [11], Rothermel et al. described an empirical study that investigated the application of program spectra. They found that for some spectra types, there was a high probablility that the spectra would exhibit a difference if there was a failure under testing. Another conclusion from [11] was that there were three types of program spectra that had nearly equivalent capability in detecting failures.

The study reported in [11] utilized seven small programs (the Siemens programs). We wished to investigate whether the results of that study might generalize to other, larger programs. This led us to perform the same experiment with a larger, industrial program with real faults.

4.2 Program Spectra

This section formally defines program spectra and outlines the different types of spectra investigated in this study. A program spectrum is developed by instrumenting the code and capturing traces of various test cases as they execute the code. The precision of the information captured in the trace depends upon which type of spectrum is generated. Figure 4.1 provides a graphic depiction of the spectra subsumption hierarchy. Table 4.1 summarizes the program spectra that we investigate. We provide an example of all spectra for two different executions of program `Sums` in Table 4.2.

4.2.1 Branch Hit Spectra

Branch Hit Spectra (BHS) are spectra that in their method of profiling contain whether or not particular branches were executed by a given test case or not. Row 1 (Branch), columns 3 and 5 (Hit) of Table 4.2 provide an example.

Abbreviation	Name	Description
BHS	Branch hit spectrum	conditional branches that were executed
BCS	Branch count spectrum	number of times each conditional branch was executed
PHS	Path hit spectrum	paths (intraprocedural, loop-free) that were executed
PCS	Path count spectrum	number of times each path (intraprocedural, loop-free) was executed
CPS	Complete path spectrum	complete paths that were executed
FRS	Fault revealing spectrum	spectra that compare outputs
ETS	Execution trace spectrum	execution trace that was produced

TABLE 4.1: A catalog of program spectra.

		Execution 1 (input is 10)		Execution 2 (input is 8, 2, 4)	
Spectrum Type	*Spectra*	Hit	Count	Hit	Count
Branch	(1,2)	Y	1	Y	1
	(3,4)	N	0	Y	2
	(3,7)	Y	1	Y	1
Path	(1,2,3,7)	Y	1	N	0
	(1,3,7), (1,2,3,4,5,6,7), (1,3,4,5,6,7)	N	0	Y	1
Complete-path	(1,2,3,7)	Y	NA	N	NA
	(1,2,3,$(4,5,6,3)^2$,7)	N	NA	Y	NA
Fault-revealing	`sum` is 0	Y	NA	N	NA
(Output)	`sum` is 6	N	NA	Y	NA
Execution-trace	(S1,S2,S3,S7)	Y	NA	N	NA
	(S1,S2,S3,$(S4,S5,S6,S3)^2$,S7)	N	NA	Y	NA

TABLE 4.2: Spectra for program `Sums` of Figure 2.1

4.2.2 Branch Count Spectra

Branch Count Spectra (BCS) are spectra that in their method of profiling contain, for each branch in a program, the number of times that branch was executed. Whereas BHS contain a boolean "true" or "false" if a branch was hit, BCS contain a nonnegative number specifying the number of times that branch was hit. Row 1 (Branch), columns 4 and 6 (Count) of Table 4.2 provide an example.

4.2.3 Path Hit Spectra

Path Hit Spectra (PHS) are spectra that in their method of profiling contain for each loop-free, intraprocedural path[1] in a program, whether that path has been executed by a given test case. Row 2 (Path), columns 3 and 5 (Hit) of Table 4.2 provide an example.

4.2.4 Path Count Spectra

Path Count Spectra (PCS) are spectra that in their method of profiling contain, for each loop-free, intraprocedural path in a program, the number of times that path has been executed by a given test case. Whereas PHS contain a boolean "true" or "false" if a path was executed, PCS contain a nonnegative number specifying the number of times each path was executed. Row 2 (Path), columns 4 and 6 (Count) of Table 4.2 provide an example.

---

[1] A loop-free, intraprocedural path is a path through a single procedure where all control stays within that given procedure and for all the statements in the procedure, each statement is executed at most once.

4.2.5 Complete Path Spectra

Complete Path Spectra (CPS) are spectra that in their method of profiling contain the entire path that is traversed as the program executes. Stated another way, CPS tracks the individual nodes visited in the control flow graph representing the program. However, only the identifiers of nodes of the control flow graph are recorded and not the text of specific statements executed. Row 3 (Complete-path), columns 3 and 5 (Hit) of Table 4.2 provide an example.

4.2.6 Fault Revealing Spectra

Fault Revealing Spectra (FRS) are spectra that compare the output between an original and a modified program. FRS differ from other spectra because instrumentation is not used to generate them. Instead, they involve comparing the output of a base and a modified program. If a base program P and a modified program P' generate different output for the same test case t, FRS will indicate this difference in output. In the case of corrective maintenance, where no specification changes have occurred, then the presence of an output difference necessarily implies the presence of a fault. If all factors involving the testing environment are held constant with respect to previous runs, this fault must have been caused by modifications. Row 4 (Fault-Revealing), columns 3 and 5 (Hit) of Table 4.2 provide an example.

FRS are not practical spectra: we use them in our study to compare with other spectra to determine the correlation between spectra differences and faults.

$$
\begin{tikzcd}
 & ETS \arrow[d] \arrow[dr] & \\
 & CPS \arrow[d] & FRS \\
 & PCS \arrow[dl] \arrow[dr] & \\
BCS \arrow[dr] & & PHS \arrow[dl] \\
 & BHS &
\end{tikzcd}
$$

FIGURE 4.1: Spectra subsumption hierarchy. The notation A $\longrightarrow$ B indicates that spectra of type A subsumes spectra of type B.

4.2.7 Execution Trace Spectra

Execution Trace Spectra (ETS) are spectra that contain the entire sequence of program statements encountered as the program executes. Row 5 (Execution-trace), columns 3 and 5 (Hit) of Table 4.2 provide an example.

4.3 Previous Empirical Study

As mentioned in Section 4.1, Rothermel et al. empirically studied the relationship between spectra and faults [11]. They discovered that ETS will *always* exhibit spectral differences for inputs that cause faults to occur. Other spectral (i.e., CPS, PCS, and BCS) differences very frequently correlate with fault occurrences. Also, the CPS, PCS, and BCS spectra less frequently display spectra differences on inputs that do not cause faults than ETS does.

Another finding reported in [11] was that CPS, PCS, and BCS spectra were nearly equivalent in their abilities to distinguish program differences. Thus, BCS would most likely be the most cost-effective of those three spectra because it requires the least overhead to collect. This conclusion comes after analyzing the costs of collecting each respective spectra.

In [11], the Siemens programs were used. The goal in this study, therefore, is to see whether similar results occur with a much larger, industrial-based subject *Space*.

4.4 The Experiment

Again, the goal of this experiment is to replicate the experiment reported in [11], but on a substantially different subject. Thus, our study uses the objectives, measures, and design used in [11]. We describe these here.

4.4.1 Objectives

We seek to investigate the following research questions:

1. Given a program P, faulty version P', and universe of inputs U for P, what correlation exists between inputs that cause P and P' to produce different spectra and inputs that reveal a fault in P? More precisely:

    (a) How often does an input $i \in U$ that causes P' to fail produce different spectra for P and P'?

    (b) How often does an input $i \in U$ that produces different spectra for P and P' cause P' to fail?

2. What are the relationships between the various spectra types, both in terms of their correlation with program-failure behavior, and in terms of their correlation with one another?

4.4.2 Measures

Rothermel et al. [11] use two measures to quantify the degree to which the presence of a spectral difference correlates with the presence of faulty behavior: *imprecision* and *unsafety*.

Imprecision is a measure of how often there is a spectral difference that is not correlated with a failure. If there is a spectral difference without a failure, then that particular spectrum is imprecise because it identified a failure when it should not have.

Unsafety is a measure of how often there is a failure that is not correlated with a spectral difference. If there is a failure without a spectral difference, then that particular spectrum is unsafe because it did not identify a failure when it should have.

Ideally, a program spectrum would be perfectly precise and perfectly safe. That would mean that if given a spectral difference there would definitely be a fault and if given a failure, there would definitely be a spectral difference.

Another goal of this experiment is to compare spectra against each other to discover which spectra are relatively more effective. So for each original program, modified program, and test pool of inputs, and each pair of spectra types S_1 and S_2, we calculated the following:

1. The number of inputs in U that cause spectral differences of type S_1.

2. The number of inputs in U that cause spectral differences of type S_2.

3. The number of inputs in U that cause spectral differences of type S_1 but not of type S_2.

4. The number of inputs in U that cause spectral differences of type S_2 but not of type S_1.

4.4.3 Subjects

This study uses the *Space* program as described in Section 3.1. Rather than use individual test suites to generate spectra, however, we used the entire test pool of 13,585 test cases. Because execution of the experiment method on each version required approximately 330 hours, we restricted our attention to 20 of the 38 faulty versions.

4.4.4 Experiment Design

For this experiment, we calculated the spectra for each [program-modified version] pair for each input of the universe of inputs (i.e., 13,585 test cases). Following this, the spectra were compared to the FRS results obtained on the program, versions and universe of inputs.

The independent variable in this experiment is the spectra: BHS, BCS, CPS, PHS, PCS, FRS, and ETS. The dependent variable measured is the set of inputs in the universe input file that revealed spectral differences between the original and modified programs. By using this data, both unsafety and precision can be calculated.

4.4.4.1 Instrumentation

We calculated BHS, BCS, PHS, PCS, CPS, FRS, and ETS spectra for each modified version *Space'* of the original *Space* program. Numerous tools were used. For generating the ETS spectra, we used the Dejavu [24] tool. For FRS spectra, we compared outputs between *Space* and *Space'* for each given test case in the universe to determine if the test case discovered a fault within the modified program. Aristotle [9] provided tools for creating the BHS, BCS, PHS, PCS, and CPS spectra. Throughout the experiment, we viewed *Space* as the "correct" version and all other *Space'* versions as ill-fated attempts to modify that version.

4.4.4.2 Experiment Method

To ensure proper replication of the experiment in [11], our experiment design follows the process used in that study. The procedure involved first running the base version on the test universe to obtain traces. Next, the paths for the base version were generated. Then each version was executed to generate their traces and paths by using the same test suite as input. Both the traces and the paths from the base version and the modified versions were subsequently utilized to generate the appropriate spectra. The spectra were then compared with each other to determine how often they differed.

4.4.5 Threats to Validity

There are threats to validity of the experiment that must be taken into account when assessing its results. There are *external* threats to validity: factors that limit the ability to generalize the results of the study to a larger set of software subjects. One external threat is that the subject program, *Space*, is a single program, and not necessarily representative of a general class of programs. This is true for all programs because currently there are not any standard sets of representative programs so any chosen program would have this threat. Similarly, the faults within the *Space* program may not be representative of the kinds of faults that occur in most programs. To reduce these threats, this study must be repeated with different subjects. This study reduces the external threat to validity of the previous study [11] because it is the same experiment with a different subject. By considering these results together with those of the earlier study, we begin the process of addressing this threat.

There are also *internal* threats to validity: influences that can affect the dependent variables without the researchers' knowledge. The main internal threat to validity is the threat of instrumentation effects. Stated another way, the software tools used to instrument the *Space* program could have unknowingly changed the actual way the program executes so that it behaves differently, and consequently bias the results of the study. To minimize internal threats to validity, we performed several validity checks on our results, including examining them for conformance with respect to the theoretical spectra hierarchy relation. (Refer to Figure 4.1 for spectra hierarchy relation.) No efforts were made, however, to control for the structure of the source programs or for the area where changes in the programs happened.

4.5 Data and Analysis

The strategy used to analyze the data generated from the experiment is to calculate both the degree of imprecision and the degree of inclusiveness for each

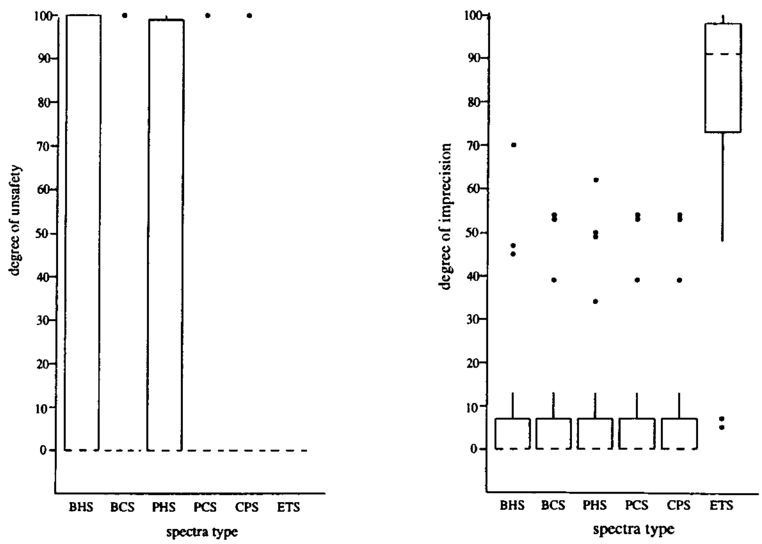

FIGURE 4.2: Boxplot graphs showing the degrees of unsafety and imprecision of spectra.

spectra type. For each spectra, the degree of imprecision is calculated for the original program, the modified program, and the universe of inputs. Likewise, for each spectra, the degree of unsafety is calculated for the original program, the modified program, and the universe of inputs.

Figure 4.2 shows boxplots presenting the degrees of unsafety and imprecision over the 20 different modified versions. The results shown in the figure are similar to those shown in [11]. The vertical axes list degrees of unsafety and precision, respectively; the horizontal axes list spectra types. In each boxplot, the dashed line represents the median of the degree of imprecision or unsafety that occurred for that spectra. The box indicates the interquartile range—the range in which the middle half of the data falls—and also indicates where those data fall with respect to the median. The "whiskers" above and/or below boxes indicate the percentages at which data above or below the interquartile range

A	B	C	D	E
Spectra (S_1-S_2)	Number of S_1 differences	Number of S_2 differences	S_1 differences not different in S_2	S_2 differences not different in S_1
BHS-BCS	46131	49994	0	3863
BHS-PHS	46131	47655	0	1524
BHS-PCS	46131	49994	0	3863
BHS-CPS	46131	49994	0	3863
BCS-PHS	49994	47655	2339	0
BCS-PCS	49994	49994	0	0
BCS-CPS	49994	49994	0	0
PHS-PCS	47655	49994	0	2339
PHS-CPS	47655	49994	0	2339
PCS-CPS	49994	49994	0	0
FRS-BHS	47308	46131	2923	1746
FRS-BCS	47308	49994	925	3611
FRS-PHS	47308	47655	2235	2582
FRS-PCS	47308	49994	925	3611
FRS-CPS	47308	49994	925	3611
FRS-ETS	47308	201711	0	154403
ETS-BHS	201711	46131	155580	0
ETS-BCS	201711	49994	151717	0
ETS-PHS	201711	47655	154056	0
ETS-PCS	201711	49994	151717	0
ETS-CPS	201711	49994	151717	0

TABLE 4.3: Comparison of spectra summarized over all modified versions, considering each for the entire input universe (271,700 inputs).

fell; however, data points at a distance of greater than 1.5 times the interquartile range are considered outliers, and represented by small circles.

The unsafety data shown in Figure 4.2 indicates that we can expect to see spectra differences when there are program failures under testing. All the spectra types have a median degree of unsafety of 0%. Furthermore, three spectra (CPS, PCS, and BCS) demonstrate a 0% degree of unsafety over the entire first, second, and third quartiles of their data. This means that CPS, PCS, and BCS spectra identified faults for *every* input that caused a fault to be executed on at least three-fourths of the modified versions. BHS and PHS, in contrast, had a broader range of unsafety results. The BHS spectra had the second and third quartiles of its boxplot span the entire percentile range from 0-100% unsafety. PHS displayed a similarly large second and third quartile, with data ranging from 0-99% unsafety. Only the ETS spectra was found to *always* be safe (i.e., 0% unsafe): for every input that exercised a fault, for every (program, modified version) pair, there was an ETS spectral difference present. No other spectra could make this claim although CPS, PCS, and BCS came relatively close in terms of displaying a small degree of unsafety.

The imprecision data shown in Figure 4.2 indicates that no spectra are perfectly precise; they all exhibit some degree of imprecision. As in [11], the ETS spectra showed itself to be the most imprecise with a median degree of imprecision of 91%. The median degree of imprecision for the other spectra (PHS, PCS, BHS, BCS, and CPS) was 0%.

The CPS, PCS, and BCS spectra display *exactly* identical behavior regarding imprecision and unsafety. Reps et al. [17] conjectured that PCS would be more adept at identifying different program behavior than BCS but the results of this experiment, similar to those found in [11], contradict that conjecture.

Table 4.3 lists the relationship between the various spectra. Column **A** lists the spectra compared; Column **B** lists the total number of inputs that cause spectra differences of type S_1; Column **C** lists the total number of inputs that cause spectra differences of type S_2; Column **D** lists the total number of inputs that cause spectra differences of type S_1 but not of type S_2;

Column **E** lists the total number of inputs that cause spectra differences of type S_2 but not of type S_1.

Figure 4.3 provides a graphic depiction of some of the data in Table 4.3. The six outer squares represent the comparison of the FRS spectra to the other five spectra, respectively, as labeled. Each such square represents the entire universe of input points over all modified versions. Within the outer squares, the lightly shaded areas indicate the percentages of input points that caused only XS-spectra differences ($XS \neq$ FRS), the medium shaded areas represent the percentages of input points that caused only FRS-spectra differences, and the darkly shaded areas represent the percentages of input points under both XS and FRS. Note that the medium shaded areas for FRS-CPS, FRS-BCS, and FRS-PCS are so negligible that they are not visible. This figure shows another view of the data given in the boxplots in Figure 4.2. For example, ETS is greatly imprecise but *safe*.

Figure 4.4 provides a more in-depth perspective of the relationship of BHS, BCS, CPS, PHS, and PCS, that displays, for each of those spectra, the number of inputs for which spectral differences existed. Again, the figure shows that CPS, PCS, and BCS demonstrate *exactly* the same behavior. Also, theoretically neither PHS nor BCS subsumes one another but empirically BCS subsumed PHS: BCS contained 2339 more spectral differences than did PHS. So whenever there was a PHS spectral difference, there was also a BCS spectral difference. Reference [11] observed a similar relationship.

Rothermel et al. [11] found PCS to never be more sensitive than BCS over the 245,087 inputs in their experiment and discovered CPS to be more sensitive than PCS on only 7 inputs. Similarly, we found in our study that CPS, BCS, and PCS had *identical* sensitivities for all 271,700 inputs. This means that on

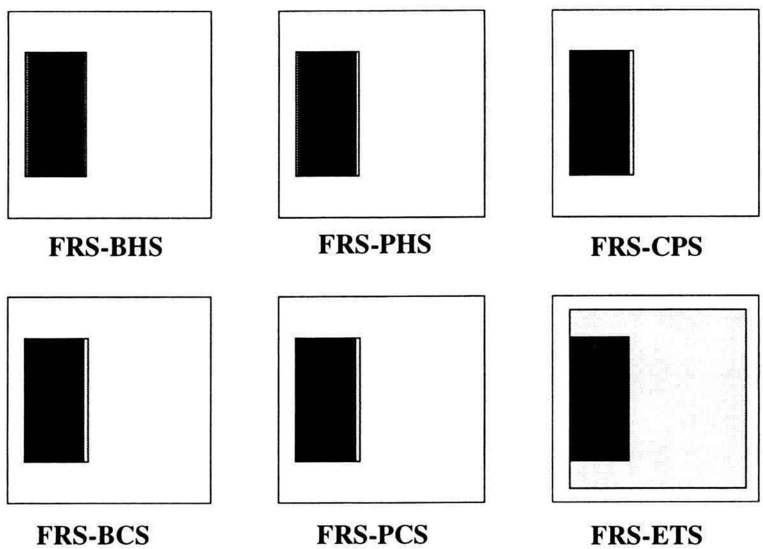

FIGURE 4.3: Graphical comparison of *FRS* with the other spectra.

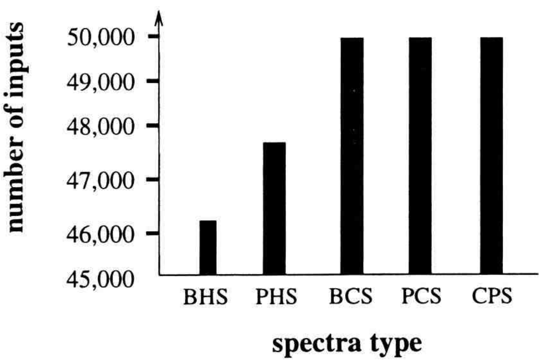

FIGURE 4.4: Comparison of CPS, BCS, PCS, PHS, and BHS, showing, for each spectra type (horizontal axis), the number of inputs for which spectra differences occurred (vertical axis).

Space as in the earlier study, PCS, BCS, and CPS effectively collapsed into one another and thus did not properly subsume one another as shown in Figure 4.1.

4.6 Conclusion

This experiment has been a replication of a previous experiment [11] with a larger, industrial scale subject, *Space*, studying the correlation between spectra and program failure behavior, along with the relationship between spectra amongst one another. While the subject differed, the results ultimately showed remarkable similarity to the previous results.

This experiment has studied spectra in one manner: running the same inputs over base, modified version pairs. Conclusions cannot be drawn as to the validity of the application of spectra, suggested in [17], for the year 2000 problem. We caution that this is only the second empirical study of this nature and more studies are needed to adequately assess spectra's applicability to software regression testing. However, it is encouraging that these results are similar to those found in [11].

There are some threats to validity in this experiment such as representativeness of the program and instrumentation effects. As a result, we took steps to minimize these threats. More experiments with different subject programs will lessen these threats.

Our results have the following implications for practice. Although ETS are the only spectra that are not unsafe, the cost required to generate them may be excessive, and they are very imprecise. The other spectra, PCS and BCS, identify faults with high frequency and cost much less to generate. Keep in mind that PCS and BCS will cost somewhat in unsafety: these spectra sometimes may allow faults to go unnoticed in the testing process. Another advantage of CPS, PCS, and BCS is that they are far less imprecise than ETS.

Between CPS, PCS, and BCS, the most cost-effective spectra is BCS because it has *exactly* the same precision as the other two spectra and costs less in program instrumentation. Profiling for BCS spectra incurs a 16% run-time overhead whereas profiling PCS incurs up to 30% [3].

Chapter 5

CONCLUSION AND FUTURE WORK

We have performed and described two experiments, investigating techniques for aiding software regression testing.

The first experiment, examining the regression test selection techniques `Pythia` and `Dejavu`, supports the conclusion that `Pythia` is often more efficient than `Dejavu` in the cases we examined. Our data also show that `Dejavu` occassionally is more precise than `Pythia`. Consequently, regression testers must assess their own particular situation to decide which RTS technique is appropriate for their situation.

The second experiment, investigating program spectra, supports conclusions found in [11]. The successful replication of that study with a larger subject program, *Space*, has helped reduce the external threat to validity for that experiment. The main conclusion from the second experiment is that, for the cases studied, BCS provides the best cost-to-collect/faults-caught ratio, among the spectra studied.

To reduce the external threats to validity and generalize the results of both experiments, these experiments must be performed again with different subject programs. Also, the first experiment should be conducted again with larger test suites and programs in order to learn what cost test suite execution must have in order for `Dejavu` and `Pythia` to be more efficient than the retest-all technique.

That future experiment could also investigate whether or not `Pythia`, which is interpreted, will be less efficient than `Dejavu`, which is a binary program.

Overall, this research has contributed to computer science in two ways. First, it has provided data to both regression test researchers and ultimately, software regression testers that will help increase the effectiveness of regression testing. Perhaps more important, however, this research helps promote an empirical approach to computer science, in which theories are tested empirically. In doing so, the research helps computer science progress further from a new discipline to a discipline more similar to well-established "hard" sciences.

BIBLIOGRAPHY

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, first edition, 1986.

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

[3] T. Ball and J.R. Larus. Efficient path profiling. In *MICRO-29*, December 1996.

[4] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, NY, 1990.

[5] J. Bible, G. Rothermel, and D. Rosenblum. A comparative study of coarse- and fine-grained safe regression test selection. Technical Report 99-60-05, Computer Science Department, Oregon State University, Mar 1999.

[6] A. Cancellieri and A. Giorgi. Array preprocessor user manual. Technical Report IDS-RT94/052, 1994.

[7] Y.F. Chen, D.S. Rosenblum, and K.P. Vo. TestTube: A system for selective regression testing. In *Proceedings of the 16th International Conference on Software Engineering*, pages 211–222, May 1994.

[8] T.L. Graves, M.J. Harrold, J-M Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *The 20th International Conference on Software Engineering*, April 1998.

[9] M.J. Harrold, L. Larsen, J. Lloyd, D. Nedved, M. Page, G. Rothermel, M. Singh, and M. Smith. Aristotle: a system for the development of program-analysis-based tools. In *Proceedings of the 33rd Annual Southeast Conference*, pages 110–119, March 1995.

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

[11] M.J. Harrold, G. Rothermel, R. Wu, and L. Yi. An Empirical Investigation of Program Spectra. In *1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 83–90, June 1998.

[12] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, pages 191–200, May 1994.

[13] H.K.N. Leung and L. White. Insights into regression testing. In *Proceedings of the Conference on Software Maintenance*, pages 60–69, October 1989.

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

[15] S.L. Pfleeger. *Software Engineering: Theory and Practice.* Prentice Hall College Division, New York, NY, first edition, 1998.

[16] R. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, New York, NY, 1987.

[17] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. *ACM Software Engineering Notes*, 6:432–439, November 1997.

[18] D. Rosenblum and G. Rothermel. A comparative study of regression test selection techniques. In *Proceedings of the 2nd International Workshop on Empirical Studies of Software Maintenance*, October 1997.

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

[20] D. Rosenblum and E. J. Weyuker. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Tranactions son Software Engineering*, 23(3):146–156, March 1997.

[21] G. Rothermel and M.J. Harrold. A safe, efficient algorithm for regression test selection. In *Proceedings of the Conference on Software Maintenance*, pages 358–367, September 1993.

[22] G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, August 1996.

[23] G. Rothermel and M.J. Harrold. Experience with regression test selection. In *International Workshop for Empirical Studies of Software Maintenance*, November 1996.

[24] G. Rothermel and M.J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[25] G. Rothermel and M.J. Harrold. Experience with regression test selection. *Empirical Software Engineering Journal*, 2, 1997.

[26] G. Rothermel and M.J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, pages 401–419, June 1998.

[27] G. Rothermel, M.J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings of Conference on Software Maintenance*, pages 34–43, November 1998.

[28] S. Schach. *Software Engineering.* Aksen Associates, Boston, MA, 1992.

[29] W.F. Tichy. Should computer scientists experiment more? *IEEE Computer*, pages 32–40, May 1998.

[30] F.I. Vokolos. *A regression test selection technique based on textual differencing.* PhD thesis, Jan 1998.

[31] F.I. Vokolos and P.G. Frankl. Pythia: A regression test selection tool based on textual differencing. In *Proceedings of the 3rd International Conference on Reliability, Quality & Safety of Software-Intensive Systems (ENCRESS '97)*, May 1997.

[32] F.I. Vokolos and P.G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Proceedings of Conference on Software Maintenance*, pages 44–53, November 1998.

[33] L. Wall and R. Schwartz. *Programming Perl.* O'Reilly and Associates, Sebastopol, CA, 1991.

[34] L.J. White, V. Narayanswamy, T. Friedman, M. Kirschenbaum, P. Piwowarski, and M. Oha. Test Manager: a regression testing tool. In *Proceedings of the Conference on Software Maintenance*, pages 338–347, September 1993.

[35] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, pages 230–238, November 1997.

[36] M.V. Zelkowitz and D.R. Wallace. Experimental models for validating technology. *IEEE Computer*, pages 23–31, September 1990.