Empirical Evaluation of the Textual Differencing Regression Testing Technique

Filippos I. Vokolos
Outsource Laboratories
Shrewsbury, NJ, USA
filip@olabs.com

Phyllis G. Frankl
Polytechnic University
Brooklyn, NY, USA
phyllis@morph.poly.edu

Abstract

Regression testing is a commonly used activity whose purpose is to determine whether the modifications made to a software system have introduced new faults. Textual differencing is a new, safe and fairly precise, selective regression testing technique that works by comparing source files from the old and the new version of the program. We have implemented the textual differencing technique in a tool called Pythia. Pythia has been developed primarily through the integration of standard, well known UNIX programs, and is capable of analyzing large software systems written in C. In this paper we present results from a case study involving a software system of approximately 11,000 lines of source code written for the European Space Agency. The results provide empirical evidence that textual differencing is very fast and capable of achieving substantial reductions in the size of the regression test suite.

1 Introduction

Most software development efforts use *testing* as the principal method to determine whether the software deviates from the specified requirements. Typically, software testing is carried out in different phases and there is a close relationship with the various phases of the life-cycle. For example, at the time of module development *unit testing* is conducted. When major software components are integrated to produce one or more of the subsystems *integration testing* takes place. Finally, when the system exists as a complete entity, the software undergoes *system testing*. Each of these testing phases varies in scope, but in theory the various test phases should complement each other and share the same objective, that is, try to uncover faults that have been created during the specification, and/or implementation of the software.

Software that has been modified, whether to fix a known fault, or to provide additional functionality, should be retested with the following objectives: (1) ensure that the new specifications have been implemented correctly; (2) establish that the modified program still works correctly on all of the test cases used to test the original program; (3) test those parts of the application that have not been tested before. The process of retesting the software to establish that the modified program still works correctly on all of the test cases used to test the original program (objective 2) is known as *regression testing*.

The traditional approach to regression testing is to exercise *all* the test cases that were used to test the software before the modifications were made. In practice, especially with large software systems, this may not be practical, primarily due to time and cost. In these situations the testing organization must decide which test cases to use in their regression testing. Typically, testing organizations employ ad hoc selection methods; consequently, the regression testing effort, when completed, does not provide a high level of confidence that indeed the code modifications did not introduce any new faults.

The subject of regression testing has received a fair amount of attention from the software testing research community, especially in recent years. Various techniques have been proposed and some have been implemented, to mechanize the process of identifying a *selective subset* from all the available test cases that should be used during regression testing.

These techniques, known as *selective regression testing techniques*, vary on both the granularity of the analysis that they perform and on the expected characteristics of the software system to be analyzed. Rothermel and Harrold have surveyed and compared these techniques in [26]. They define several attributes that a good selective regression testing technique should have, including safety, precision, and efficiency. Roughly speaking a technique is safe if it selects all of the test cases from the original test suite that could potentially reveal faults in the modified program and is precise if it does not select too many additional, unnecessary test cases. These terms are defined more carefully below. There is generally some trade-off between safety, precision, and efficiency. In addition, they argue that for a technique to be useful it should be possible to build a robust tool that works on real industrial programs.

Two of the most promising techniques are the ones developed by Rothermel and Harrold [25], and Chen,

Rosenblum, and Vo [6]. The technique developed by Rothermel and Harrold is based on the idea of creating control flow graphs (CFGs) to represent and compare the old and the new versions of a program. It is a safe technique, capable of selecting a fairly precise subset of the test cases. It has been implemented in a prototype research tool called **DejaVu**. The technique developed by Chen, Rosenblum, and Vo is based on the idea of detecting modified code entities such as functions, variables, types, and preprocessor macros. It has been implemented in a tool called **TestTube**, which is less precise than DejaVU, but more robust in analyzing large software systems. Ball has recently explored the limitations of techniques based on control-flow analysis and has proposed several techniques of this nature, including one that is marginally more precise than the DejaVu technique at comparable cost, and another that is somewhat more precise, but more expensive [2].

We have previously developed a different approach to selective regression testing, called *textual differencing* and a tool, Pythia, based on that approach [29][30]. Our motivation was to create a safe selective regression testing tool that balances precision, efficiency, and the ability to support large software systems. Textual differencing works by comparing the program text from source files, rather than using an abstract representation of the program. Pythia, which runs on the UNIX environment and which can be used to analyze software systems written in C [15], has been implemented by integrating standard, well known, UNIX programs.

Although our analytical evaluation of Pythia suggested favorable precision and efficiency performance characteristics, we did not have empirical results to substantiate our analysis. We have recently conducted an empirical evaluation of Pythia using a program written for the European Space Agency. In this paper we present the results of this evaluation. These results provide evidence that indeed Pythia is very fast and capable of achieving substantial reductions in the sizes of test suites.

The paper is organized as follows: Section 2 defines the terms used in this paper and provides some background on the selective regression testing problem. Section 3 describes Pythia, its components, architecture, and how it works. Section 4 describes the experiment and presents the results. Section 5 concludes the paper with a summary and plans for future work.

2 Terminology and background

A *test case*, denoted by t_i, is an element of the input domain. It is associated with the output that results from the execution of the program, which we denote as $P(t_i)$. We refer to the set of test cases $T = \{t_1, t_2, ...\}$ used to test the program P as the *test suite* for P.

A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

The execution of P with input t_i results in the execution of a sequence of program statements, and by extension, basic blocks, called the *execution trace*. Saving the entire execution trace sometimes requires a large amount of space, especially for programs that iterate loops many times. Instead, we work with the set of blocks that have been executed when program P is executed on test t_i, which we denote by $ET(P, t_i)$. $ET(P, t_i)$ does not record the order or number of times those blocks were executed.

The ideal goal of *selective regression testing* is as follows: Given a program P, a modified version of that program which we denote by P', and T, determine through static analysis the subset $T'_{ideal} = \{t \in T \mid P'(t) \neq P(t)\}$. To be cost effective, the time required for the analysis should be less than the time required to execute and verify the results for the test cases in T but not in T'_{ideal}. Since the outputs of P and P' are the same on those test cases in $T - T'_{ideal}$, it is not necessary to execute P' on any of those test cases[1].

In general — in the absence of dynamic information from the execution of P or P' — there is no algorithm to compute the set T'_{ideal}, for arbitrary P and P'. This follows from the undecidability of the famous *halting problem* [7], which states that there is no algorithm to determine whether an arbitrary program P halts on a given input.

One realistic alternative towards a solution for the selective regression test problem is to obtain useful information from the execution of P and combine this information with further static analysis to obtain T', an approximation of T'_{ideal}. This is the approach taken by all known code-based selective regression testing techniques and it is also the approach taken by textual differencing.

A statement is considered *modified* if it is replaced (or *changed*) with a new set of statements, if it is *deleted*, or if a new set of statements have been *inserted* immediately following that statement.

A selective regression testing technique is *safe* if it selects (from the available test suite) all test cases t, such that $P(t) \neq P'(t)$. That is, T' is a safe subset of T iff $(P(t) \neq P'(t) \Rightarrow t \in T')$. If P and P' are executed on *identical operating environments*[2] and T' is a safe subset of T, then exe-

1. If the specification has changed, it is necessary to assure that the outputs of those test cases satisfy the new specification, but it may be possible to determine this from archived test results or by reasoning about the nature of the specification changes without re-executing the test cases.
2. We use the term *operating environment* to denote all the things that may influence the execution of the program, such as h/w architecture, operating system, environment variables, interrupts, etc.

cuting T' on P' will detect any failures introduced by the modifications on P that T would have detected [6][26]. In this paper we assume that P and P' are executed on identical operating environments.

Precision, defined as $1 - (|T'|/|T|)$, measures the extent to which a technique omits test cases that do not produce different outputs in P and P'. Precision is an important attribute of regression testing techniques [26]. A technique which is not very precise will tend to select test cases that should not really be selected. The less precise the technique is, the closer the size of the selected test suite will be to the size of the original test suite.

Given the constraint that the analysis has to be performed in time less than that required to execute and verify the results for the test cases in T but not in T', the objective towards a solution for the regression test problem is a safe technique with good trade-off between low cost and high precision.

3 Pythia

Pythia is a selective regression testing tool capable of analyzing software systems written in C. It has been implemented by integrating the following three UNIX programs: `cc`, the C language compiler, `pretty`, a beautifier for C programs, and `diff`, the general purpose file comparison program.

To determine the regression test suite, Pythia maintains a history of the basic blocks executed by each test case and compares the source files of the old and new versions of the program to identify the modified program statements. Each modified source statement is mapped to the appropriate basic block and all test cases that executed that specific basic block are selected. The source-to-source comparison is done with `diff`. To ensure that `diff` will produce useful results, Pythia first transforms the source files into *canonical* form using a prettyprinter. A prettyprinter, or program beautifier, is a program that takes as input source code and prints the code with aesthetically appropriate indentations and line breaks [20][27]. Execution tracing is done by instrumenting the source code using the C compiler. The components that make up Pythia are described below.

`kform` is a script around the program `pretty`; it transforms source files into canonical form. `instr` is a script around the C compiler; it instruments source files in canonical form to produce execution traces. The program being tested is executed via the program `xqt`, which maintains a history of test cases along with the basic blocks executed by each test case. `txt` is the program that compares the old with the new canonical files (by calling `diff`), analyzes the differences reported by `diff`, and computes the set of all test cases that have exercised modified statements.

To do this, `txt` associates differences in source code lines reported by `diff` 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 (e.g. modification of a simple statement in the middle of a sequence of statements forming a block). In such cases, Pythia selects all test cases that execute that block. In more complicated situations, a textual difference may potentially 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 modification execute block b.

For example, under some circumstances, changes to the "then" part of an "if" statement may cause selection of all test cases that reached the "if" condition, regardless of whether they cause it to evaluate to "true" or "false".

In addition, Pythia can handle changes in declarations. In a C file, declarations may appear anywhere in the file. The scope of a declaration that appears outside a function is the set of functions that follow the declarations. A modification to a declaration statement that appears outside a function will cause Pythia to select all the test cases that have traversed the initial basic block of any function that follows the declaration. statement in the file. A modification to a declaration statement that appears inside a function will cause Pythia to select all the test cases that have traversed the basic block defined immediately following the declaration statement. For further details on Pythia refer to [29]. For a detailed description of the algorithms used to analyze the output produced by `diff`, refer to [30].

To briefly illustrate how Pythia works, we consider a small C program that raises a floating point number to an integer power, using Dijkstra's algorithm. The example is based on a C function written by I.S. Dunietz [8].

The program, called `power`, consists of two files: `main.c` and `power.c`. Each file contains a single C function. The old versions of `main.c` and `power.c` are shown in Figure 1 and Figure 3, respectively. The new versions are shown in Figure 2 and Figure 4. For the benefit of space we show these files in their canonical forms.

```
 1    extern double power();
 2
 3    extern double atof();
 4
 5    extern int atoi();
 6
 7    extern void printf();
 8
 9    extern void exit();
 10
 11   main(argc, argv)
 12   int argc;
 13   char *argv[];
[14]  {
 15     double x;
 16     int n;
 17
 18     if (argc != 3)
 19     {
[20]         exit(0);
 21     }
[22]   x = atof(argv[1]);
 23     n = atoi(argv[2]);
 24     printf("power(%.1f, %d) =\n", x, n);
 25     printf("%g\n\n", power(x, n));
 26   }
```

Figure 1: *Old* version of `main.c`

```
1    extern double power();
2
3    extern double atof();
4
5    extern int atoi();
6
7    extern void printf();
8
9    extern void exit();
10
11   main(argc, argv)
12   int argc;
13   char *argv[];
14   {
15     double x;
16     int n;
17
18     if (argc != 3)
19     {
20         printf("Illegal number of arguments\n");
21         exit(1);
22     }
23     x = atof(argv[1]);
24     n = atoi(argv[2]);
25     printf("power(%.1f, %d) =\n", x, n);
26     printf("%g\n\n", power(x, n));
27   }
```

Figure 2: *New* version of `main.c`

```
 1    double power(x, n)
 2    double x;
 3    register int n;
[4]   {
 5      int recip, sgn;
 6      double y;
 7
 8      if (n < 0)
 9      {
[10]        recip = 1;
 11         n = -n;
 12     }
 13     else
 14     {
[15]        recip = 0;
 16     }
[17]   sgn = 1;
 18    if (x < 0.0e0)
 19    {
[20]        x = -x;
 21    }
[22]   for (y = 1.0e0; n > 0; --n)
 23    {
[24]        while (n % 2 == 0)
 25         {
[26]            x *= x;
 27             n /= 2;
 28         }
[29]        y *= x;
 30    }
[31]   if (recip != 0 && y != 0.0e0)
 32    {
[33]        return (sgn * 1.0e0 / y);
 34    }
 35    else
 36    {
[37]        return (sgn * y);
 38    }
 39   }
```

Figure 3: *Old* version of `power.c`

```
1    double power(x, n)
2    double x;
3    register int n;
4    {
5      int recip, sgn;
6      double y;
7
8      if (n < 0)
9      {
10         recip = 1;
11         n = -n;
12     }
13     else
14     {
15         recip = 0;
16     }
17     sgn = 1;
18     if (x < 0.0e0)
19     {
20         x = -x;
21         if (n % 2 == 1)
22         {
23             sgn = -1;
24         }
25     }
26     for (y = 1.0e0; n > 0; --n)
27     {
28         while (n % 2 == 0)
29         {
30             x *= x;
31             n /= 2;
32         }
33         y *= x;
34     }
35     if (recip != 0 && y != 0.0e0)
36     {
37         return (sgn * 1.0e0 / y);
38     }
39     else
40     {
41         return (sgn * y);
42     }
43   }
```

Figure 4: *New* version of `power.c`

The initial version of the program was tested using the following set *T* of test cases:

	input value	output	
$t_{1:}$	-5.0, 2	25	
t_2:	-3.0, 3	27	incorrect
t_3:	2.0, 0	1	
t_4:	1.0, 4	1	
t_5:	0.0, -1	0	

Test case t_2 (negative base with a positive, odd, exponent) revealed a fault. For all other test cases the program computed the correct values. Using the line number associated with a program statement as the basic block identifier, the execution trace for each function was as follows:

ET(main,*T*) = {14, 22}

ET(power, t_1) = {4, 15, 17, 20, 22, 24, 26, 29, 31, 37}

ET(power, t_2) = {4, 15, 17, 20, 22, 24, 26, 29, 31, 37}

ET(power, t_3) = {4, 15, 17, 22, 31, 37}

ET(power, t_4) = {4, 15, 17, 22, 24, 26, 29, 31, 37}

ET(power, t_5) = {4, 10, 17, 22, 24, 29, 31, 37}

Assume we modify the program (as highlighted in Figure 2) to address the following:

- Correct the `exit(0)` statement on line 20 in the function `main()`.
- Print an error message when `power` is invoked with incorrect number of arguments.
- Correct the fault revealed by t_2.

The comparison of the old and the new versions of the file `main.c` will produce the following `diff` output:

```
20c20,21
< exit(0);
---
> printf("Illegal number of arguments\n");
> exit(1);
```

The output indicates that the statement on line 20 was replaced by two different statements. The statement on line 20 defines a basic block and as a result Pythia will select all the test cases that executed that specific basic block. Since none of the test cases executed the basic block on line 20, Pythia will return the null set.

The comparison of the old and the new versions of the file `power.c` will produce the following `diff` output:

```
20a21,24
> if (n % 2 == 1)
> {
>       sgn = -1;
> }
```

The output indicates that new code was inserted immediately following the statement on line 20. Since line 20 of `power.c` also defines a basic block, Pythia will select all the test cases that executed that specific basic block. The execution trace indicates that only two test cases, namely t_1 and t_2, went through that basic block and as a result Pythia will return these two test cases as the content of the regression test suite.

4 Empirical evaluation

To evaluate the effectiveness and efficiency of textual differencing, as it has been implemented by Pythia, we performed a case study in which Pythia was used to analyze thirty-three (33) different versions of a C program. The goals of this study were to measure the percent reduction in test suite size achieved by Pythia on a group of large industrial programs with real modifications, and to measure the analysis time and any other overhead required to achieve this test set reduction. In this section we present the results of this case study.

As our test subject we selected a program developed by Ingegneria Dei Sistemi (Pisa - Italy) for the European Space Agency. The program, called ORACOLO2, allows the user to describe the configuration of an array of antennas using a high-level language instead of writing the complete list of element positions and excitations. The language, which was specifically developed for ORACOLO2, is called ADL (Array Definition Language).

ORACOLO2 consists of three different subsystems. The *parser* subsystem performs syntactic analysis of the input sentences and sets up the internal data structures that describe the array. The *computation* subsystem computes and assigns values to array parameters which have been omitted by the user, executes consistency checks on user specified parameters, and computes the position and excitation of each array element. The *formatting* subsystem prepares and outputs the data in the required format.

ORACOLO2 is written in C and was developed within the Microsoft Visual C++ 1.5 environment. Its relevant characteristics are:

Number of modules (i.e., C functions): 135

Total number of source lines (w/ comments): 11,640

Size of executable (in bytes): 139,264

We selected ORACOLO2 as our test program for two important reasons: First, there is a well documented change

history of the program, and second we had access to a test case generator, capable of generating "random" ADL input files. Although we would have preferred to have access to test cases developed specifically for and used with each version of the program, the test case generator simplified substantially our task of testing a program that we knew very little about. ORACOLO2 has been used by Pasquini, Crespo, and Matrella in a recent case study of various reliability growth-models [22].

The documented change history of ORACOLO2 is as follows: At different times during testing and the operative use of the program, thirty-three (33) different faults were discovered and recorded. These faults, shown in Table 1, were classified by the developers and/or Pasquini, Crespo, and Matrella according to the *IEEE 1044 - Standard Classification for Software Errors, Faults, and Failures*:

Fault Type & Subtypes	# of Faults
Logic omitted or incorrect	
Forgotten case or steps	2
Unnecessary function	1
Missing condition test	4
Checking wrong variable	2
Computational Problems	
Equation insufficient or incorrect	10
Interface incorrect or incomplete	
Module mismatch	3
Data handling problems	
Data initialized incorrectly	1
Data accessed or stored incorrectly	10
TOTAL	33

Table 1: Types of faults detected in the test program.

Each fault was corrected and a new version of the program was created and released for subsequent testing or operative use, as appropriate. Thus, each of version of the program, contains all the corrections made to address a particular fault.

To correct the thirty-three different faults, twenty-four different functions had to be modified. One function was modified for three separate versions (i.e., the function was modified to address three different faults), eight functions were modified for two separate versions, and the remaining fifteen functions were modified for a single version each.

4.1 Approach

A single copy of the ORACOLO2 source files contains the source code for all thirty-three versions of the program. To create the executable image corresponding to the i^{th} version of the program, the source files are compiled with the macro `#define` i^{th} turned on. Since Pythia requires separate source files for each version of the program, the first step was to "extract" and store in different directories the source code for each version. To do so we executed the C language preprocessor, `cpp`, each time with the corresponding `#define` macro turned on. The source files representing the *gold* (i.e., last) version of the program were created by preprocessing the files with all of the `#define` macros turned off.

Having the source files for the gold and the other thirty-three different versions of the program, we proceeded with the following steps, as required by Pythia:

i. Transformed the source files of the gold version into canonical form, using the program `pretty`.
ii. Created the executable image for the gold version, using the C compiler with the `-a` option.
iii. Executed the gold version with the available test suite, using the program `xqt`.
iv. Transformed the source files of each version into canonical form, using `pretty`.
v. Analyzed the source files representing the gold version against the source files representing the other thirty-three different versions to determine the contents of the regression test suite for each version.

This process is exactly opposite from the way we do things in real life, where we start with a faulty version, modify it and retest it, to eventually create the correct version. The approach we took was strictly out of convenience and had to do with the way the software is bundled. (To create a specific version, the gold version has to be compiled with the appropriate `#define` macro turned on.)

We ran our experiment six different times. Each experiment was performed with a separate test suite. Each test suite was randomly generated with the available test case generator. For our initial pilot experiment, E_0, the test suite contained 10 different test cases. The rest of the experiments, i.e., E_1 - E_5, consisted of test suites that contained 100 different test cases each.

4.2 Results

We conducted the experiments on a Sun SPARC 20 running SunOS 4.1.4. The analysis revealed that the gold version in canonical form consisted of 6,752 lines and 1,274 different basic blocks. The size of the instrumented version of the executable was approximately 18% larger than the corresponding non-instrumented version (139Kb vs. 163Kb). As expected, the instrumented version of the executable was slower than the corresponding non-instru-

mented version. For example, for 1 test case, execution time — as reported by the UNIX program `time` in "real time" — went from `0m0.05s` to `0m0.30s`. For 10 test cases, execution time went from `0m0.78s` to `0m2.35s`. Finally, for 100 test cases, execution time went from `0m11.5s` to `0m27.0s`. In some development environments, this program instrumentation may be added for other reasons, such as checking code coverage or looking for frequently executed code segments which can potentially be tuned for better performance. However, in an environment in which tests are normally run without instrumentation, this added overhead may affect the decision as to whether it is cost-effective to use (any) selective regression testing technique.

The number of test cases selected for each of the 33 versions of our initial experiment, E_0 (10 different test cases) and of the other five experiments, i.e., E_1 - E_5 (100 different test cases) are shown below:

	E_0	E_1	E_2	E_3	E_4	E_5
1:	10	96	99	97	96	97
2:	2	26	33	24	25	33
3:	10	96	100	97	97	97
4:	10	96	99	97	96	97
5:	2	26	33	24	25	33
6:	10	96	100	97	97	97
7:	1	5	2	6	6	5
8:	0	1	2	2	4	3
9:	4	57	56	56	56	63
10:	1	19	18	27	26	24
11:	3	25	21	24	14	15
12:	1	2	0	0	0	0
13:	1	3	4	11	11	12
14:	8	86	87	89	85	89
15:	8	75	75	74	70	82
16:	1	4	6	5	4	5
17:	8	61	64	62	58	71
18:	1	2	0	0	0	0
19:	2	24	23	27	29	18
20:	0	0	2	3	2	2
21:	0	0	2	3	2	2
22:	1	4	4	2	2	3
23:	2	7	7	7	6	8
24:	3	7	8	5	8	8
25:	4	57	56	56	56	63
26:	2	18	21	19	18	16
27:	1	4	6	5	4	5
28:	8	86	88	89	88	91
29:	10	100	100	100	100	100
30:	8	86	88	89	88	91
31:	3	16	12	13	17	17
32:	7	63	67	61	53	67
33:	1	4	4	2	2	3

4.3 Analysis of Results

A quick look at the results reported for all the experiments shows that in a large number of cases the regression test suite selected by Pythia substantially reduced the number of test cases that must be re-executed during regression testing. For instance, focusing on the results reported in E_1 - E_5, in almost 40% of the versions (13/33), Pythia reduced the size of the regression test suite by at least 90% on average, while in almost 50% of the versions (16/33), a reduction of at least 80% was reported. Figure 5 shows the number of ORACOLO2 versions that achieved certain levels of test case reduction.

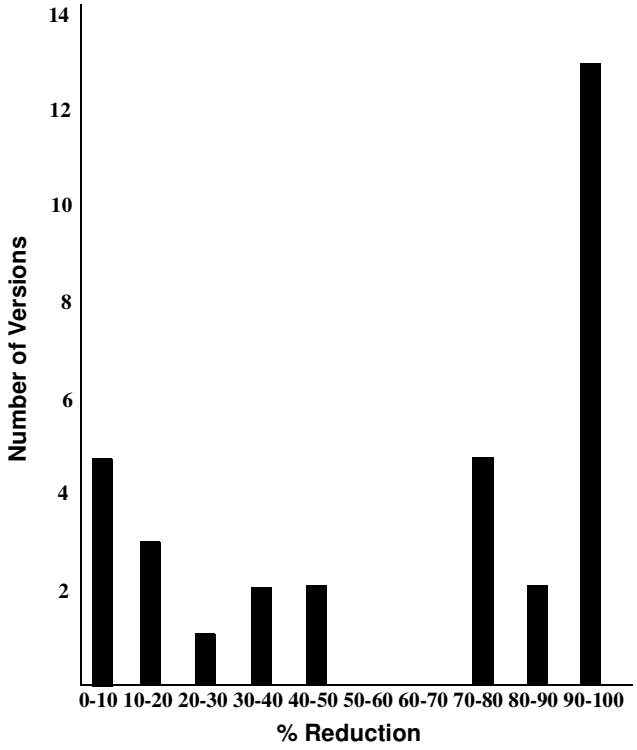

Figure 5: Reduction in the number of test cases

It is very encouraging that Pythia was able to analyze without any assumptions, or problems, a large number of different versions of a real C program and in many cases, it reported substantial reductions in the number of the test cases required for regression testing. Furthermore, the analysis of the source was done very quickly. It took Pythia less than 6 seconds to analyze ORACOLO2, a time that for many large industrial strength systems can be safely characterized as negligible.

With respect to the time required for the analysis, it appears, based on our data, that textual differencing is much faster than the technique developed by Rothermel and Harrold [25]. Data presented by Rothermel [24] in connection with a study similar to the one described in this paper indicates the following: On a Sun SPARC 10, the

graph walk technique required 16.8 seconds to create control-flow graphs for a C program consisting of 21 functions and 512 non-commented lines of code. In contrast, on a Sun SPARC 20, textual differencing required 5 seconds to transform 11,640 commented lines of code into canonical form, which is approximately one third of the time for over 20 times the size. Furthermore, in approximately equal time (5.35 sec. *vs*. 5 sec.), Pythia was able to analyze a program approximately 13 times the size of the program analyzed by the graph walk technique (6,752 *vs*. 512). Clearly, the slower performance of the graph walk technique has to do, in part, with the fact that DejaVu was executed on a slower processor (SPARC 10 *vs*. SPARC 20).

4.4 Threats to Validity

The fact that we have used a small number of test suites with randomly generated test cases raises legitimate concerns about the interpretation and generalizibility of these results. More and better controlled empirical evaluations will be required before we can claim with confidence the overall effectiveness of textual differencing.

The data presented and analyzed in this section are focused on test case reduction and the cost required for the analysis of the program. For a selective regression testing technique to be beneficial, ultimately, it must achieve a reduction in the time required to execute and verify the results of the selected test cases vis a vis the initial test suite [19]. As it turned out, our case study was not a good vehicle to address this issue. The time required to execute a test case with ORACOLO2 is *very* small, about `0m11.5s` for 100 test cases. Considering the increased cost to execute a test case due to instrumentation (`0m27s` for 100 test cases), even a 100% reduction in the number of test cases would make the technique unattractive, unless instrumentation is already being done for other reasons.

For most versions, the variance among the five runs E_1 to E_5 was fairly small, but for some it was larger. In some versions Pythia selected zero test cases, indicating that the original test suites did not cover the portions of *P* that contained those modifications. These issues call into question the extent to which the nature of the original randomly generated test suites affect the results. Further experimentation, perhaps using larger test suites or test suites chosen to satisfy some adequacy criterion might shed light on this issue.

The results of this case study -- substantial reduction in test suite size, fairly small execution time for the program analysis, and ability to handle analyze a large industrial system are encouraging. However, it should be remembered that these 33 programs all came from the same development team and project. Further studies using different subjects would be useful.

Lastly, we note that, although we believe that Pythia is almost as precise as DejaVu, we did not investigate this issue directly, nor did we perform head-to-head comparison of the efficiency of Pythia with any other tools. Further empirical studies, comparing cost and precision of various tools and verifying that they are safe would be useful.

5 Conclusion

We have presented the results from a case study that we recently completed to evaluate the effectiveness and efficiency of the textual differencing regression testing technique, as it has been implemented by the tool Pythia.

The results are very encouraging. They point out that Pythia can quickly analyze software systems written in C and can be effective in reducing the set of test cases that should be used for regression testing. We need, and plan, to do further empirical evaluations to better understand the performance and potential benefits that Pythia may have when used with arbitrary, large, industrial software systems.

6 Acknowledgements

We would like to thank Alberto Pasquini and Paolo Matrella for providing us with a copy of ORACOLO2. Gregg Rothermel made several useful suggestions that have improved the readability and accuracy of the paper. Frankl's work was supported in part by NSF grant CCR-9206910.

7 References

[1] AGRAWAL, H., J.R. HORGAN, E.W. KRAUSER, and S.A. LONDON. "Incremental Regression Testing", *Proc. Conf. on Software Maintenance 1993*, pp. 348-357, [Sep. 1993].

[2] BALL, T. "On the limit of Control Flow Analysis for Regression Test Selection", *Proc. Int. Symposium on Software Testing and Analysis (ISSTA 1998)*, pp. 134-142, [Mar. 1998]

[3] BATES, S., and S. HOROWITZ. "Incremental Program Testing Using Program Dependence Graphs", *Proc. 20th ACM Symp. on Principles of Programming Languages*, 9(9), pp. 384-396, [Jan. 1993].

[4] BENEDUSI, P., A. CIMITILE, and U. DECARLINI. "Post-maintenance Testing Based on Path Change Analysis", *Proc. Conf. on Software Maintenance 1988*, pp. 352-361, [Oct. 1988].

[5] BINKLEY, D. "Semantics Guided Regression Test Cost Reduction", *IEEE Trans. on Software Engineering*, SE-23(8), [Aug. 1997].

[6] CHEN, Y-F., D.S. ROSENBLUM, and K-P. VO. "TestTube: A System for Selective Regression Testing", *Proc. 16th Int. Conf. on Software Engineering*, pp. 211-220, [May 1994].

[7] DAVIS, M.D., R. SIGAL, and E.J. WEYUKER. Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science. 2nd ed. San Diego, CA.: Academic Press, [1994].

[8] DUNIETZ, I.S. "The addmon Family of Tools", *AT&T Bell Laboratories Technical Memorandum*, 103122000-930426-01TMS, [May 1993].

[9] FISHER, K., F. RAJI, and A. CHRUSCICKI. "A Methodology for Retesting Modified Software", *Proc. National Telecommunications Conf. 1981*, pp. B6.3.1-B6.3.6, [Nov. 1981].

[10] GUPTA, R., M.J. HARROLD, and M.L. SOFFA. "An Approach to Regression Testing using Slicing", *Proc. Conf. on Software Maintenance*, pp.299-308, [Nov. 1992].

[11] HARROLD, M.J., and M.L. SOFFA. "An Incremental Approach to Unit Testing During Maintenance", *Proc. Conf. on Software Maintenance 1988*, pp. 362-367, [Oct. 1988].

[12] HARTMANN, J., and D.J. ROBSON. "Approaches to Regression Testing", *Proc. Conf. on Software Maintenance 1988*, pp. 368-372, [Oct. 1988].

[13] ———. "Techniques for Selective Revalidation", *IEEE Software*, 7(1):31-36, [Jan. 1990].

[14] HUNT, J.W., and M.D. MCILROY. "An Algorithm for Differential File Comparison", Computer Science Technical Report 41, *Bell Laboratories*, [1975].

[15] KERNIGHAN, B.W., and D.M. RITCHIE. The C Programming Language. Prentice-Hall, Englewood Cliffs, NJ, [1978].

[16] LASKI, J., and W. SZERMER. "Identification of Program Modifications and its Application in Software Maintenance", *Proc. Conf. on Software Maintenance 1992*, pp. 282-290, [Nov. 1992].

[17] LEUNG, H.K.N., and L. WHITE. "A Study of Integration Testing and Software Regression at the Integration Level", *Proc. Conf. on Software Maintenance 1990*, pp. 290-301, [Nov. 1990].

[18] ———. "Insights into Testing and Regression Testing Global Variables", *Journal of Software Maintenance*, 2, pp. 209-222, [Dec. 1990].

[19] ———. "A Cost Model to Compare Regression Test Strategies", *Proc. Conf. on Software Maintenance 1991*, pp. 201-208, [Oct. 1991].

[20] OPPEN, D.C. "Prettyprinting", *ACM Trans. Programming Languages and Systems*, 2(4), pp. 465-483, [Oct. 1980].

[21] OSTRAND, T.J., and E.J. WEYUKER. "Using Data Flow Analysis for Regression Testing", *Proc. 6th Annual Pacific Northwest Software Quality Conf.*, pp. 233-247, [Sep. 1988].

[22] PASQUINI, A., A.N. CRESPO, and P. MATRELLA. "Sensitivity of Reliability-Growth Models to Operational Profile Errors *vs* Testing Accuracy", *IEEE Trans. on Reliability*, R-45(4), pp. 531-540, [Dec. 1996].

[23] ROSENBLUM, D.S. "Towards a Method of Programming With Assertions", *Proc. 14th Int. Conference on Software Engineering*, pp. 92-104, [May 1992].

[24] ROTHERMEL, G. "Efficient, Effective Regression Testing Using Safe Test Selection Techniques", Ph.D. diss., Clemson University, [May 1996].

[25] ROTHERMEL, G., and M.J. HARROLD. "A Safe, Efficient Regression Test Selection Technique", *ACM Trans. on Software Engineering and Methodology*, 6(2), pp. 173-210, [Apr. 1997].

[26] ———. "Analyzing Regression Test Selection Techniques", *IEEE Trans. on Software Engineering*, SE-22(8), pp. 529-551, [Aug. 1996].

[27] RUBIN, L.F. "Syntax-Directed Pretty Printing — A First Step Towards a Syntax-Directed Editor", *IEEE Trans. on Software Engineering*, SE-9(2), pp. 119-127, [March 1983].

[28] SHERLUND, B., and B. KOREL. "Modification Oriented Regression Testing", *Conf. Proc. Quality Week 1991*, pp. 1-17, [May 1991].

[29] VOKOLOS, F.I. "A Regression Test Selection Technique Based on Textual Differencing", Ph.D. diss., Polytechnic University, [Jan. 1998].

[30] VOKOLOS, F.I., and P.G. FRANKL. "Pythia: A regression test selection tool based on textual differencing", *Proc. 3rd Int. Conf. on Reliability, Quality & Safety of Software-Intensive Systems (ENCRESS '97)*, [May 1997].

[31] WHITE, L.J. et al. "Test Manager: A Regression Testing Tool", *Proc. Conf. on Software Maintenance 1993*, pp. 338-347, [Sep. 1993].

[32] YAU, S.S., and Z. KISHIMOTO. "A Method for Revalidating Modified Programs in the Maintenance Phase", *COMPSAC '87: The 11th Annual Int. Computer Software and Applications Conf.*, pp. 272-277, [Oct. 1987].

ERROR: undefined
OFFENDING COMMAND: ---------------BC41D7E72AD571EBC98BA910--

STACK: