Pythia: A regression test selection tool based on textual differencing

F. I. Vokolos
AT&T Labs
Murray Hill, NJ, USA, +1-908-582-7127, email: filip@att.com

P. G. Frankl
Polytechnic University
Brooklyn, NY, USA, +1-718-260-3870, email: 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. For many large, complex, software systems the retest all strategy is not practical: the resources required to reexecute and verify all available test cases (i.e., time and human effort) are prohibitive. Ad hoc methods are not desirable, as they can compromise the reliability of the regression test activity and consequently the reliability of the software system being tested. In this paper we present a new technique for selecting regression test cases based on the modifications that have been made on the program. The technique, which is based on the idea of directly comparing source files from the old and the new version of the program, has been implemented in a tool called Pythia. A novel characteristic of Pythia, which is capable of analyzing large software systems written in C, is that it has been implemented primarily through the integration of standard, well known, UNIX[1] programs.

Keywords

Regeression testing, selective regression testing techniques, regression test selection.

1. UNIX is a registered trademark of X/Open and licensed exclusively through X/Open Co. Ltd.

1 INTRODUCTION

Most software systems that have been developed have used *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.

The development of a typical software system requires a number of iterations between module development and system testing and before the system is released to its user community. Once the system is released, then the software enters the maintenance phase of the life-cycle. During the maintenance phase the system will undergo many changes. Some of these changes fix known faults, while others provide additional functionality. The amount of modification made to the code to support these changes varies greatly from simple statement changes to a complete rewrite of the system.

Our experience as developers and testers, as well as statistics referenced in the literature [22], indicate that the likelihood of introducing faults while making modifications is substantial. Software that has been modified, whether to fix a known fault, or to provide additional functionality, should be retested with the following objectives: (i) ensure that the new specifications have been implemented correctly, (ii) establish that the modifications made to the code have not introduced any new faults, and (iii) test those parts of the application that have not been tested before. The process of retesting the software to determine that the modifications have not introduced any new faults is known as *regression testing*. In theory, regression testing should 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 is not 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 high level of confidence that indeed the code modifications did not introduce any new faults.

Over the years, various techniques have been proposed, and some have been implemented, to mechanize the process of identifying the test cases that should be included in the regression test suite. These techniques vary on both the level of analysis being performed and on the expected characteristics of the software system being analyzed.

In this paper we discuss a new technique that we have developed to select test cases for regression testing. We call this technique *textual differencing* because it works by comparing the program text from source files, rather than using an abstract representation of the program. We have implemented this technique in a tool called Pythia which runs on the UNIX environment and which can be used to analyze software systems written in the programming language C [17]. A novel characteristic of Pythia is that it has been implemented by integrating standard, well known, UNIX programs.

The paper is organized as follows: Section 2 defines the terms used in this paper. Section 3 briefly overviews recent work in the area of test case selection for regression testing. Section 4 discusses the textual differencing technique and the implementation of Pythia. It also provides a small example that illustrates how textual differencing works. Section 5 analyzes textual differencing by using accepted criteria for the analysis of such techniques. We conclude in section 6 with a summary and our plans for future work.

2 TERMINOLOGY AND BACKGROUND

A *program P* is a collection of one or more functions. Each function consists of a collection of statements $<S_1, \ldots, S_n>$. P' denotes a modified version of the program *P*. We use the terms *old version* to refer to *P* and *new version* to refer to *P'*. A *basic block* is a sequence of consecutive statements $<S_i, \ldots, S_m>$ with the property that control enters at the beginning statement S_i and may leave only at the very last statement S_n.

A *test case*, denoted by t_i, is an identifiable set of inputs accepted by the program along 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*.

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*. $ET_B(P(t_i))$ denotes the execution trace of basic blocks for test case t_i.

Selective regression testing refers to the strategy of retesting the modified program using some subset of the available test suite. The test cases chosen by a selective regression test technique form the *selected test suite*.

A test case is considered to be *modification-traversing* [25] if it executed code that was subsequently changed, inserted into *P* (to create *P'*), or deleted from *P*. A selective regression testing technique is *safe* if it selects (from the available test suite) all the modification-traversing test cases. *Precision* measures the extent to which a technique omits test cases that do not produce different outputs in *P* and *P'*.

If *P* and *P'* are executed on identical operating environments[1] and *T'* is a safe subset of *T*, then executing *T'* on *P'* will detect any failures introduced by the modifications on *P* that *T* would have detected. In the rest of this paper we assume that *P* and *P'* are executed on identical operating environments.

3 OVERVIEW OF RELATED WORK

The subject of selective regression testing has received a fair amount of attention from the software testing research community, especially in recent years [1], [2], [3], [4], [6], [9], [10], [11], [12], [13], [18], [19], [20], [21], [22], [24], [26], [29], [31]. Rothermel and Harrold have surveyed and compared these techniques in [25]. In what follows in this section we discuss two promising techniques capable of analyzing large software systems.

Rothermel and Harrold [24] have developed a regression test selection technique that is based on the idea of creating *control flow graphs* (CFGs) to represent, and compare, *P* and *P'*.

---

1. 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, exceptional interrupts, etc.

The nodes in the CFG contain actual program statements. During the execution of *P*, a list of all the edges traversed by each test case is maintained. The CFGs are compared by simultaneously traversing the nodes of each graph and looking for differences in either (i) the contents of a node, or (ii) the contents of succeeding nodes. When differences are detected, the test cases that have traversed the edges associated with these nodes are selected.

The Rothermel and Harrold technique supports both intraprocedural and interprocedural analysis and is capable of detecting, with good precision, modification traversing test cases. Two different prototype tools, **DejaVu1** (for intraprocedural analysis) and **DejaVu2** (for interprocedural analysis) have been developed to analyze C programs. The authors have used these prototype tools on a large software system with encouraging results. However, as they point out, they were not able to instrument, or run their implementation, on about 15% of the procedures.

Chen, Rosenblum, and Vo [6] have developed a regression test selection technique based on the idea of detecting modified code entities such as functions, variables, types, and preprocessor macros. Test cases that have traversed modified code entities form the selected test suite. This technique has been implemented in a tool called **TestTube**, which has been developed around existing analysis tools, namely **app** (the Annotation Preprocessor for C [23]) and **CIA** (the C Information Abstractor [7]).

The guiding principle in developing the modified entities technique has been to reach a balance between efficiency and precision. The result has been a technique that is capable of fully analyzing large software systems in C and which is considered to be the most efficient safe regression test selection technique available [25]. However, the analysis performed is fairly coarse-grained and as a result the technique is not as precise as the one developed by Rothermel and Harrold.

4 PYTHIA

Pythia is a UNIX-based regression test selection tool that can analyze software systems written in C. It implements an analysis technique that we call *textual differencing*. The differentiating characteristic of textual differencing (from other analysis techniques for the selective regression test problem) is that **it compares source files** from the old and the new versions of the program, using a general purpose text comparison tool, in order to determine statement differences that may potentially affect the contents of the selected test suite. We feel that this is an important characteristic in light of the fact that in the past, the idea of comparing source files, using a general purpose file comparison tool, in applications that require a comparison between two versions of a program (such as the selective regression test problem) has been viewed by researchers as inadequate [25], [30]. In summary, the characteristics of Pythia are:

- It selects a safe regression test suite.
- It can be used on stand alone C functions, as well as on software systems composed of many C functions. That is, it supports both *intra*procedural and *inter*procedural analysis.
- It has been implemented primarily through the integration of existing, widely used, UNIX programs.
- The comparison between the two versions, *P* and *P’*, is done by the well known UNIX program `diff`, directly on the program text, rather than on an abstract representation of the program.

- Instrumentation, for determining the execution trace of *P*, is done directly by the C compiler, during module compilation.
- In principle, it can be easily extended to support other popular programming languages, such as C++.

The major UNIX programs that have been integrated to implement Pythia are: **`cc`**, the C language compiler, **`pretty`**, a beautifier for C programs, and **`diff`**, the general purpose file comparison program. Pythia consists of the following stand-alone programs: **`kform`**, **`instr`**, **`xqt`**, and **`txt`**. The program `txt` is written in Perl [28]; the other three programs are written in KornShell [5]. A data flow diagram showing these components is shown in Figure 1. The functionality of these programs and a high-level description on how Pythia works is as follows:

i. The source files for the old version of the program are converted -- using the program **`kform`**-- into a *canonical form*. **`Kform`** is a script around the program `pretty`, the C program beautifier.
ii. The canonical files, i.e., the source files in canonical form, are instrumented and compiled using the program **`instr`**. Instrumentation is used to maintain a basic block execution trace for *P*. **`Instr`** is a script around `cc`, the C compiler.
iii. 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.
iv. After the development of the new program has been completed, the new source files are also converted into canonical files with the program **`kform`**.
v. The program **`txt`** compares the old with the new canonical files, by calling the UNIX program `diff`, and analyzes the differences, as reported by diff, to determine the set of all test cases that have exercised modified statements.

The concept of a canonical form, the idea of using the C compiler for instrumentation, and the comparison of source files with the program `diff` are the essence of the technique and the tool. In what follows we discuss these in more detail.

4.1 The Canonical Form

In general, it is non trivial to compare the program text from two different versions of a program and get useful information regarding actual statement differences [14], [30]. Blank lines, comment lines, stylistic differences, such as multiple statements on a single line, are some of the obstacles. Figure 2 illustrates some of these differences.

To compare two source files, in a way that we can capture essential statement differences, we must assume that these files have been written using consistent syntactic and stylistic guidelines. Although it is unreasonable to expect that the person(s) developing and/or modifying a program will adhere to such guidelines, it is possible to mechanically transform the source code to satisfy such guidelines. Program beautifiers, or pretty-printers, have been used extensively, especially by software projects with large development teams to get all the source code into a consistent, pretty, style.

Pythia uses an existing C program beautifier to bring the source files into canonical form. The canonical form serves two very important purposes: First, it helps filter out irrelevant details that may hinder the comparison of the two programs. Second, because of the indentation of the program text, it allows us to use the text as an abstract representation of the program

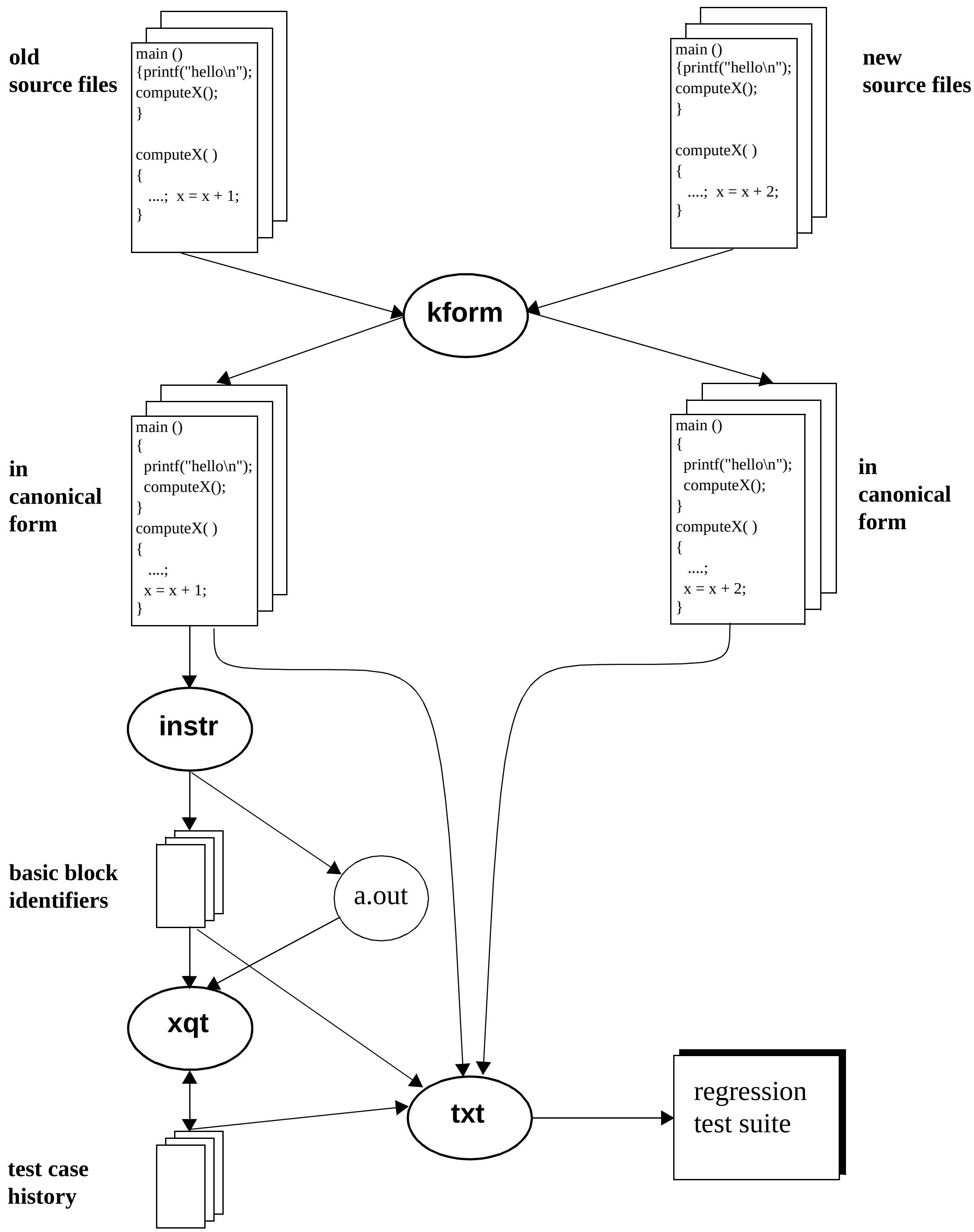

Figure 1 Pythia data-flow architecture.

during the analysis phase. We consider a source file to be in *canonical form* if it satisfies the following conventions:

1. There is a single statement on a given line.
2. There are no comment lines.
3. Blank lines (other than the ones inserted by the C program beautifier) are removed.
4. There are no split lines.
5. The text has been indented using the guidelines of structured programming.

Figure 3 shows the canonical form of the, seemingly different, C programs in the files `old.c` and `new.c` (Figure 2). The canonical form of all the C constructs (i.e., selection statements, iteration statements, and jump statements) is described in [27].

```
/* old.c: A sample C program */

main ()
{
  int x;

  scanf("%d", &x);if(x==10)
  { printf ("x is 10\n"); }
}
```

```
/*
 * new.c: The same program
 * written in different style.
 */

main ()
{
  int  x;


  scanf ("%d", &x);

  if (x == 10)
  {
      printf ("x is 10\n");
  }
}
```

Figure 2 Two seemingly different programs.

```
 1    main()
[2]   {
 3      int x;
 4
 5      scanf("%d", &x);
 6      if (x == 10)
 7      {
[8]         printf("x is 10\n");
 9      }
[10]  }
```

Figure 3 The canonical form of the files `old.c` and `new.c`. (Line numbers in square brackets denote initial statements of basic blocks.)

4.2 Code Instrumentation

Code instrumentation is a well known technique for tracing the execution of program entities, such as statements, predicates, etc. One form of code instrumentation (the one typically used in connection with software testing tools) works by inserting new statements in the code to act as counters, during the execution of the program.

There are commercially available, as well as research, code instrumentation tools that one can choose depending on the scope of the application. Typically, code instrumentation tools will modify the source code. The modified source will then be compiled and linked to create the executable program. Development and testing teams (especially the ones working on software systems that are critical in nature) are concerned about using instrumented code when in system testing and/or in production. The concern is centered around efficiency and, at times, around reliability, as the executable comes from modified source.

Pythia takes a different approach with respect to instrumentation. Rather than using a tool that modifies the source code, it uses the compiler directly to instrument and compile the source code in a single invocation. Instrumentation is done through the option of the C compiler that maintains a list of all the basic blocks traversed during the execution of the program. In theory, the instrumentation done by the compiler should represent a more desirable approach, as one would expect that it would result in more efficient, and perhaps even more reliable, executable code.

In Figure 3, and all subsequent examples, we have used square brackets around the line numbers to denote the initial statement of each basic block. Although, ultimately, Pythia looks at modified statements, knowing which basic blocks have been executed provides enough information to do further analysis at the statement level. Actually, keeping track of basic block execution traces is much more desirable than keeping track of statement execution traces as the latter can be many times larger.

4.3 Source File Comparison

Given the source files in canonical form, Pythia uses the program `diff` to get a list of the statement differences between the old and new versions of the program. Knowing how these two versions differ at the statement level, and having basic block execution traces, Pythia is able to identify all the modification-traversing test cases.

Given two text files, `diff` performs a line-by-line comparison and produces a prescription using the operations **c**hange, **a**dd, and **d**elete that will convert one file into another. For example, the comparison between `switch1.c` and `switch2.c`, shown in Figure 4, will produce the following `diff` output:

```
14a15,16
>        printf("CASE 200\n");
>        break;
16c18
<        printf("CASE 200 or 300\n");
---
>        printf("CASE 300\n");
26d27
<     printf("end_of_program\n");
```

```
 1    main ()
[2]   {
 3      int x;
 4
 5      scanf("%d", &x);
 6      if (x < 1000)
 7      {
[8]         printf("x is less than 1000\n");
 9          switch (x)
 10         {
 11         case 100:
[12]            printf("CASE 100\n");
 13             break;
 14         case 200:
 15         case 300:
[16]            printf("CASE 200 or 300\n");
 17             break;
 18         default:
[19]            printf("NOT 100,200,or 300\n");
 20             break;
 21         }
 22     }
 23     else
 24     {
[25]        printf("x is .GE. to 1000\n");
 26         printf("end_of_program");
 27     }
[28] }
```

```
1    main ()
2    {
3      int x;
4
5      scanf("%d", &x);
6      if (x < 1000)
7      {
8          printf("x is less than 1000\n");
9          switch (x)
10         {
11         case 100:
12             printf("CASE 100\n");
13             break;
14         case 200:
15             printf("CASE 200\n");
16             break;
17         case 300:
18             printf("CASE 300\n");
19             break;
20         default:
21             printf("NOT 100,200,or 300\n");
22             break;
23         }
24     }
25     else
26     {
27         printf("x is .GE. to 1000\n");
28     }
29   }
```

Figure 4 The files `switch1.c` and `switch2.c`.

Each operation reported by `diff` is of the form:

n1,[n2] operation n3,[n4]

context_line(s)

where *n1[,n2]* represents ranges of lines in the first file, *n3[,n4]* represents ranges of lines in the second file, *operation* is one of {**a**dd | **c**hange | **d**elete}, and *context_line(s)* are all the lines affected in both files. The lines prefixed with the less-than sign (<) pertain to the first, or in our case old, file. The lines prefixed with the greater-than sign (>) pertain to the new file. For example, the first operation reported by the command `$diff switch1.c switch2.c`, **`14a15,16`**, indicates that the lines `printf("CASE 200\n");` and `break;` should be added after line 14 in the file `switch1.c`. The second operation, **`16c18`**, indicates that the 16th line in `switch1.c` `printf("CASE 200 or 300\n");` should be replaced with the line `printf("CASE 300\n");`. Finally, the operation, **`26d27`**, says that the 26th line of the file `switch1.c` `printf("end_of_program\n");` should be deleted.

The effect that the operations **a**dd, **c**hange, and **d**elete have on the statements and keywords of the various C constructs (i.e., selection statements, iteration statements, and jump statements) has been studied by the authors and appears in [27].

In analyzing the `diff` output, Pythia processes each operation, along with its *context_line(s)*, separately. Before processing these operations, Pythia uses the algorithm **`ComputeEnclosingBasicBlock`**, discussed in section 4.4, to identify the basic block associated with a given statement. In the context of textual differencing, the **a**dd operation is treated slightly different from the **c**hange and **d**elete operations. In what follows, we discuss the semantics of each operation, with point of reference the first (or old) source file, and how we deal with each operation.

The operation add

The operation **a**dd is specified as:

*n1**a**n3[,n4]*

context_line(s)

and the meaning of the operation is: *Add the context_line(s) immediately following line n1*.

When the **a**dd operation is reported by `diff`, all the test cases that have executed that basic block associated with the statement *n1* are selected. In processing the **a**dd operation, Pythia uses the assignment of basic blocks to statements as it was computed by the algorithm **`ComputeEnclosingBasicBlock`**. In some special cases, in order to increase its precission, Pythia overwrites some of these assignments. That is, it associates a statement with a new basic block number. These special cases are:

i. When new statements are inserted after the left curly brace (`{`) that follows the statements: `if(...)`, `while(...)`, and `for(...)`.
ii. When new statements are inserted after the right curly brace (`}`) that encloses the *then* clause of an `if-else` construct.
iii. When new statements are inserted after the right curly brace (`}`) that encloses the *else* clause of an `if-else` construct.
iv. When new statements are inserted after a `case` statement of the `switch` construct.

In each of these cases, the basic block used is the basic block assigned to the next statement.

The operation change

The operation **c**hange is specified as:

*n1[,n2]**c**n3[,n4]*

context_line(s)

and the meaning of the operation is: *Replace all the lines specified by the range n1[,n2] with all the lines specified by the range n3[,n4] of the second (or "new") source file*.

Unlike the **a**dd operation, the **c**hange operation may involve multiple statements. All of these statements are specified in *context_line(s)* and are prefixed with the less-than sign (<). Since it is possible that two different statements may belong in two different basic blocks, each statement in the range *n1[,n2]* is processed separately. All the test cases that have traversed the basic block associated with each of the statements in the range *n1,n2* are selected.

The operation delete

The operation **d**elete is specified as:

*n1[,n2]**d**n3*
context_line(s)

and the meaning of the operation is: *Delete all the lines specified by the range n1[,n2]*.

The **d**elete operation is similar to the **c**hange operation, in that they both specify multiple lines that are modified in the first file. As a result, the **d**elete operation is handled the same way as the **c**hange operation.

4.4 Program Statements and Basic Blocks

As a general purpose text comparison tool, `diff` doesn't provide any information regarding the relationship between program statements and basic blocks. This relationship is essential for textual differencing, since on the one hand there exist basic block execution traces (from the instrumentation and execution phases) and on the other statement differences. To associate modified statements to basic blocks Pythia uses the algorithm **`ComputeEnclosingBasicBlock`**, a summary of which is shown in Figure 5.

```
ComputeEnclosingBasicBlock ()
{
    // INPUTS:      An array which contains the source file in canonical form.
    //              An array which contains the initial statement of each basic block.
    // OUTPUTS:     An array with the basic block number for each statement.
    // VARIABLES: level:     the indentation level of a statement.
    //            monitor:   the line # of the initial statement of a basic block.

    Get the first monitor;
    Mark all the statements of the program with the first monitor;
    While there are more monitors to process
    {
       Get the next monitor;
       Go to the statement pointed to by monitor;
       Mark that statement with the monitor;
       Let λ be the level of that statement;
       If (line n-1 is a "{" @ level λ-1 &&
           line n-2 is the keyword "else" @ level λ-1)
             Mark the lines n-1 and n-2 with the monitor;
       If (line n-1, is a label starting on column 1)
             Mark the line n-1 with the monitor;
       Mark all following statements @ level == λ with the monitor
       until either  a statement @ level == λ-1 |
                     end_of_function (i.e., } in the 1st column) |
                     the EOF is found;
       If (the statement @ level == λ-1 is the "}" of an else clause
          Mark the "}" with the same monitor as the "else" keyword;
    }
}
```

Figure 5 A summary of the algorithm **ComputeEnclosingBasicBlock**.

The algorithm is actually invoked before processing the output from `diff`. With the use of the indentation of the program text, which is part of the canonical form, in a single pass, it associates each statement in the file with the appropriate basic block.

In essence, after the execution of **`ComputeEnclosingBasicBlock`**, Pythia "knows" which basic blocks have been modified and it can then easily select (by "looking" at the basic block execution traces) all the test cases that have executed modified basic blocks.

4.5 An Example

We now consider an example to illustrate the textual differencing technique. The example we use is based on a C function written by I.S. Dunietz to raise a floating point number to an integer power, using Dijkstra's algorithm. The function `power()` appeared as part of the documentation for the **addmon** family of tools [8].

The program, called `power`, consists of two files: `main.c` and `power.c`.; each file contains one function. The old version of the program is shown in Figure 6. The new version is shown in Figure 7. For the benefit of space we assume that the first step of textual differencing (i.e., bringing the source files into canonical form) has been completed and show these files in their canonical forms.

The initial version of the program was tested using the set of test cases $T = \{t_1, t_2, t_3, t_4, t_5\}$. The input and output values for each test case are shown in the table below:

id:	input value	output value	
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 revealed a fault for the case of a negative base with a positive, odd, exponent. For all other test cases the program computed the correct values.

The basic block execution trace for each function was as follows:

$ET_B(\text{main}(T)) = \{14, 22\}$

$ET_B(\text{power}(\{t_1\}) = \{4, 15, 17, 20, 22, 24, 26, 29, 31, 37\}$

$ET_B(\text{power}(\{t_2\}) = \{4, 15, 17, 20, 22, 24, 26, 29, 31, 37\}$

$ET_B(\text{power}(\{t_3\}) = \{4, 15, 17, 22, 31, 37\}$

$ET_B(\text{power}(\{t_4\}) = \{4, 15, 17, 22, 24, 26, 29, 31, 37\}$

$ET_B(\text{power}(\{t_5\}) = \{4, 10, 17, 22, 24, 29, 31, 37\}$

The power program was modified to address the following:

i. Correct the `exit(0)` statement on line 20 in the function `main()`.
ii. Provide an error message whenever the program is invoked with incorrect number of arguments.
iii. Correct the fault revealed by t_2.

The modified and new statements are highlighted in Figure 7.

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

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

In analyzing this output the program **`txt`** determined that, since the statement on line 20 was replaced by two different statements, it has to select all the test cases that went through that statement. The statement on line 20 defines a basic block and as a result **`txt`** selected all the test cases that executed the basic block on line 20. Since none of the test cases appear in the execution trace of that basic block, **`txt`** returned the null set.

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

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

In analyzing this output the program **`txt`** determined that, since new code was inserted immediately following the statement on line 20, it has to select all the test cases that went through that statement. The statement on line 20 defines a basic block and as a result **`txt`** selected all the test cases that executed the basic block on line 20. The execution trace indicates that only two test cases, namely t_1 and t_2, went through that basic block.

As a result of this analysis, Pythia returned the following set of test cases:

$$\text{Selected Test Suite} = \varnothing \cup \{t1, t2\}$$

```c
extern double power();

extern double atof();

extern int atoi();

extern void printf();

extern void exit();

main(argc, argv)
int argc;
char *argv[];
{
    double x;
    int n;

    if (argc != 3)
    {
        exit(0);
    }
    x = atof(argv[1]);
    n = atoi(argv[2]);
    printf("power(%.1f, %d) =\n", x, n);
    printf("%g\n\n", power(x, n));
}
```

main.c

```c
double power(x, n)
double x;
register int n;
{
    int recip, sgn;
    double y;

    if (n < 0)
    {
        recip = 1;
        n = -n;
    }
    else
    {
        recip = 0;
    }
    sgn = 1;
    if (x < 0.0e0)
    {
        x = -x;
    }
    for (y = 1.0e0; n > 0; --n)
    {
        while (n % 2 == 0)
        {
            x *= x;
            n /= 2;
        }
        y *= x;
    }
    if (recip != 0 && y != 0.0e0)
    {
        return (sgn * 1.0e0 / y);
    }
    else
    {
        return (sgn * y);
    }
}
```

power.c

Figure 6 Old version of the program `power`.

```
extern double power();

extern double atof();

extern int atoi();

extern void printf();

extern void exit();

main(argc, argv)
int argc;
char *argv[];
{
  double x;
  int n;

  if (argc != 3)
  {
      printf("Illegal number of arguments\n");
      exit(1);
  }
  x = atof(argv[1]);
  n = atoi(argv[2]);
  printf("power(%.1f, %d) =\n", x, n);
  printf("%g\n\n", power(x, n));
}
```

main.c

```
double power(x, n)
double x;
register int n;
{
  int recip, sgn;
  double y;

  if (n < 0)
  {
      recip = 1;
      n = -n;
  }
  else
  {
      recip = 0;
  }
  sgn = 1;
  if (x < 0.0e0)
  {
      x = -x;
      if (n % 2 == 1)
      {
          sgn = -1;
      }
  }
  for (y = 1.0e0; n > 0; --n)
  {
      while (n % 2 == 0)
      {
          x *= x;
          n /= 2;
      }
      y *= x;
  }
  if (recip != 0 && y != 0.0e0)
  {
      return (sgn * 1.0e0 / y);
  }
  else
  {
      return (sgn * y);
  }
}
```

power.c

Figure 7 New version of the program `power`.

5 ANALYSIS OF TEXTUAL DIFFERENCING

Rothermel and Harrold have developed a framework for evaluating selective regression testing techniques. The framework consists of the following four categories: *inclusiveness*, *precision*, *efficiency*, and *generality*. Rothermel and Harrold have used this framework to analyze and compare known selective regression testing techniques [25]. In what follows we use the Rothermel and Harrold framework to analyze the textual differencing technique.

Inclusiveness: Textual differencing is a *safe* selective regression testing technique. That is, it will select all modification-traversing test cases. To see this consider the following:

`Diff` will identify all statements that are different between *P* and *P'*. The analysis of the `diff` output will consider each statement that appears in *context_line(s)*, and which is prefixed with the less-than sign (<), separately to determine the associated basic block. Consequently, at the end of the analysis test cases will be selected for all modified statements.

Precision: Textual differencing, like all other known techniques, is not 100% precise for arbitrary programs. This means that textual differencing, in addition to the modification-traversing test cases that will select, it may also select a number of test cases that are not modification-traversing.

The imprecision is due to the following: Like the techniques of Rothermel and Harrold, and Chen, Rosenblum, and Vo, textual differencing considers statement differences without any further semantic analysis. As a result, changes that do not affect in any way the behavior of the program (e.g., `while(0==0)` vs. `while(1==1)`) will be flagged and all test cases that have traversed the modified statements will be selected. Second, on a small number of occasions the current implementation of textual differencing will opt for efficiency and safety over precision [27]. Finally, the `diff` program, in its attempt to report the minimum number of line changes necessary to convert one file into the other, may include statements in the operations it prescribes that have not been modified. These statements will be analyzed and all test cases that traversed these statements will be selected.

Efficiency: As we mentioned in section 4, textual differencing consists of four major, distinct, steps, which are carried out in sequence: (i) the transformation of the original source files of *P* into canonical form, (ii) the instrumentation of *P*'s canonical files, (iii) the transformation of the original source files of *P'* into canonical form, and (iv) the source file comparison.

Steps (i) and (ii) have time complexity linear in the size (i.e., number of statements) of *P*, thus $\Theta(|P|)$. Both of these steps are performed during the testing phase of *P* and thus they do not consume any of the time available for regression testing.

Steps (iii) and (iv) are performed during regression test time. Step (iii) has time complexity linear in the size of *P'*, thus $\Theta(|P'|)$. The time complexity of the comparison of the source files, in step (iv), is dominated by the time required for `diff`, which in the worst case has time complexity $O(|P|*|P'|*\log|P|)$ [15]. In practice, `diff` has been used extensively on large software systems in connection with version control programs with acceptable performance. We feel comfortable, that in practice, the computational cost of textual differencing will be reasonable.

Generality: Textual differencing handles all forms of code modifications: insertion, deletions, and modifications of statements. It works equally as well on both intraprocedural and interprocedural regression testing. Although the technique has been implemented for C programs, in principal it can be easily extended to programs written in languages for which there exists tools to perform basic block instrumentation and to transform the original source into canonical form.

6 CONCLUDING REMARKS

We have presented a safe regression test selection technique which is based on the idea of comparing source files to determine statement differences between two versions of a program.

We have developed a tool called Pythia that implements the technique. Pythia is capable of fully analyzing large software systems written in C. Analytical studies [27] show that the technique is fairly precise, when compared to other available techniques. We believe the precision combined with the fact that Pythia has been developed by integrating exiting, well known, UNIX programs will make it attractive to development and testing organizations.

Our experience in using Pythia with small programs has been very encouraging. We are planning to conduct empirical studies using Pythia on large software systems to determine its overall effectiveness in practice. If the results from these studies indicate that indeed Pythia is a valuable tool in selecting a regression test suite, then we'll port Pythia on other languages, most likely C++. In the mean time, we are working to increase the precision of the technique by carefully examining the effect of likely modifications on various C constructs, such as the one specified by the `switch` statement.

7 ACKNOWLEDGMENTS

The work on Pythia has been inspired by the work of Yih-Farn Chen, David Rosenblum and Kiem-Phong Vo on TestTube. David Rosenblum, Phong Vo, and Larry Wehr made several important observations and suggestions during the early phases of this work.

8 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] 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].

[3] Benedusi, P., A. Cimitile, and U. De Carlini. "Post-maintenance Testing Based on Path Change Analysis", *Proc. Conf. on Software Maintenance 1988*, pp. 352-361, [Oct. 1988].

[4] Binkley, D. "Using Semantic Differencing to Reduce the Cost of Regression Testing", *Proc. Conf. on Software Maintenance*, pp. 41-50, [Nov. 1992].

[5] Bolsky, M.I., and D.G. Korn. `The Korn Shell Command and Programming Language`. Prentice-Hall, Inc., Englewood Cliffs, NJ, [1989].

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

[7] Chen, Y-F., M. Nishimoto, and C.V. Ramamoorthy. "The C Information Abstraction System", *IEEE Transactions on Software Engineering*, SE-16(3), pp. 325-334, [March 1990].

[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] Horwitz, S. "Identifying the Semantic and Textual Differences between Two Versions of a Program", Computer Science Technical Report #895, Univ. of Wisconsin, [Nov. 1989].

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

[16] Hunt, J.W., and T.G. Szymanski. "A Fast Algorithm for Computing Longest Common Subsequences", *Comm. ACM*, 20(5), pp.350-353, [May 1977].

[17] Kernighan, B.W., and D.M. Ritchie. The C Programming Language. Prentice-Hall, Englewood Cliffs, NJ, [1978].

[18] 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].

[19] 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].

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

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

[22] 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].

[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., and M.J. Harrold. "A Safe, Efficient Regression Test Selection Technique", *Ohio State University Technical Report*, OSU-CISRC-4/96-TR25, to appear in: *ACM TOSEM*.

[25] ———. "Analyzing Regression Test Selection Techniques", to appear in: *IEEE Transactions on Software Engineering*, 1996.

[26] Sherlund, B., and B. Korel. "Modification Oriented Regression Testing", *Conf. Proc. Quality Week 1991*, pp. 1-17, [May 1991].

[27] Vokolos, F.I. "A Regression Test Selection Technique Based on Textual Differencing", Ph.D. diss., Polytechnic University, (in preparation).

[28] Wall, L., and R.L. Schwartz. Programming perl. O'Reilly & Associates, Inc., Sebastopol, CA, [Jan. 1991].

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

[30] Yang, W. "Identifying Syntactic Differences Between Two Programs", *Software — Practice and Experience*, 21(7), pp. 739-755, [July 1991].

[31] 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].