The Use of Program Profiling in Software Testing

Thomas Reps
University of Wisconsin

Abstract. This paper describes new techniques to help with testing and debugging, using information obtained from path profiling. A path profiler instruments a program so that each run of the program generates a path spectrum for the execution—a distribution of acyclic path fragments that were executed during that run. Our techniques are based on the idea of comparing path spectra from different runs of the program. When different runs produce different spectra, the spectral differences can be used to identify paths in the program along which control diverges in the two runs. By choosing input datasets to hold all factors constant except one, the divergence can be attributed to this factor. The point of divergence itself may not be the cause of the underlying problem, but provides a starting place for a programmer to begin his exploration.

1. Introduction

Testing software is a difficult problem. In general, there is always the possibility that another input will expose an error not uncovered by the tests carried out so far, and thus it is impossible to know if enough testing has been carried out. Although there is no way to surmount this fundamental difficulty, testing is an extremely important—and costly—aspect of software development.

Recently, Reps et al. proposed a new class of software-testing tools [10] that make use of information obtained from path profiling. These tools make use of the following principle:

> A path profile provides a "spectrum" of the paths that were executed during a given run of a program, and provides a behavior signature for the program when executed on a particular dataset. When different runs of a program produce different path spectra, the spectral differences can be used to identify paths in the program along which control diverges in the two runs. By choosing input datasets to hold all factors constant except one, any such divergence can be attributed to this factor. The point of divergence itself may not be the cause of the underlying problem, but provides a starting place for a programmer to begin his exploration.

This paper describes how this principle can be exploited in a variety of ways to detect unusual behavior in programs. The principle offers new perspectives on testing, on the task of creating test data, and on what tools can be created to support program testing. This approach to testing is a new variant of white-box testing, which we have termed "I/B testing" [10], for "Input/Behavior" testing, by analogy with I/O testing. In contrast to I/O testing, I/B testing can reveal possible problems—by finding path-spectrum differences—even when the output of an execution run is correct. We believe that the path-spectrum-comparison technique holds the promise of providing a useful adjunct to conventional methods for testing whether programs are functioning properly (and debugging them when they are malfunctioning).

One application of the technique is in the "Year 2000 Problem" (or "Y2K problem", for short), *i.e.*, the problem of fixing computer systems that use only 2-digit year fields in date-valued data. In this context, path-spectrum comparison provides a heuristic for identifying paths in a program that are good candidates for being date-dependent computations.

This work was supported in part by the National Science Foundation under grant CCR-9625667, and by the Defense Advanced Research Projects Agency (monitored by the Office of Naval Research under contract N00014-97-1-0114).

Author's address: Computer Sciences Department, University of Wisconsin, 1210 W. Dayton St., Madison, WI 53706, USA.
E-mail: reps@cs.wisc.edu; WWW: http://www.cs.wisc.edu/~reps/

Portions of this paper are excerpted from [10]. The Wisconsin Alumni Research Foundation is in the process of seeking patent protection for the ideas described herein.

Note that the idea of comparing path spectra to identify possible execution errors is a completely different use of path profiling in program testing from another use that has been proposed for path profiles in program testing, namely as a criterion for evaluating the coverage of a test suite [15,8,3,11].

The remainder of the paper is organized into five sections: Section 2 provides background about the Y2K problem, which furnishes several examples of problems for which path-spectrum comparison is a useful technique. Section 3 describes the use of run-time profiling to locate paths in programs that are potentially problematic. Section 4 discusses the application of path-spectrum comparison to a variety of software-testing problems. Section 5 describes the implementation of a tool based on these ideas, as well as some of the results of our preliminary experiments with the tool. Section 6 presents a few final remarks.

2. The Year 2000 Problem

Because many computer programs use only two digits to record year values in date-valued data, they may process a year value of 00 as 1900 in cases where 2000 was intended. If the intended value is 2000—such as when 00 represents the value of the current year in a computation performed after the calendar rolls over on January 1, 2000—then a faulty computation may be carried out. Because computations can involve dates in the future, the phenomenon can occur well before the calendar rolls over on January 1, 2000. For example, if the (approximate) age of someone born in 1956 were calculated for January 1, 2000, he would appear to be $00 - 56 = -56$ years old! If the program tries to use the value -56 to index into a life-expectancy table, the program will either fetch a bogus life-expectancy value or quit with an error (depending on whether the run-time system catches “index-out-of-bounds” errors). In both cases, the system functions improperly. In general, such behavior can have serious—even life-threatening—consequences. This problem and a variety of other date-related problems that will show up with increasing frequency around January 1, 2000 are known collectively as the “Year 2000 Problem”.

In addition to the rollover problem with two-digit year fields, the phrase “Year 2000 Problem” has come to mean a whole host of date-related problems that will eventually crop up, many of which strike around the turn of the millennium. For example, leap years come every four years, except for centuries, except for centuries divisible by 400. Thus, the year 2000 is, in fact, a leap year. However, some programs implement the exception, but not the exception to the exception. Such a bug could cause havoc in financial transactions (*e.g.*, by causing failures in computer-driven trading) and military maneuvers (*e.g.*, by causing logistical planning failures). UNIX systems are also subject to date-representation rollover problems, most of which occur later in the 21st century.[1]

For both date-representation rollover problems and leap-year bugs, it is necessary to find the code that declares and manipulates date-valued variables, rewrite it, and test the modifications. Unfortunately, dates are hidden in programs. “Date” is not a data-type in most programming languages, and so heuristics must be developed for identifying the locations where date-valued data is manipulated. Even when a language does have a “date” data-type, there is nothing to forbid programmers from creating or encoding “raw” dates that are embedded in data of other data types, such as character strings.

The Y2K problem is in large part a management problem; however, there are serious technical problems as well, including program-analysis methods for determining the sites at which date-manipulation code occurs, code- and data-transformation algorithms, post-renovation testing, and the technical challenges of arranging for renovated and unrenovated systems to interoperate. The techniques described in this paper are relevant to two of these problems: (i) determining the sites at which date-manipulation code occurs, and (ii) post-renovation testing.

Because the leverage that tools for the Y2K problem can provide is limited by their accuracy for locating the places in a piece of code where dates are employed, the date-location issue is crucial to the creation of effective tools for correcting date-manipulation problems. Two techniques for locating dates are used in present commercial products:

[1] Overflow in the UNIX *time* function occurs on Tuesday, January 19, 2038 at 03:14:08 UTC.

(1) Some date-manipulation sites can be identified by the places where a program makes certain calls to the operating system, for example, to retrieve the current date.
(2) Other date-manipulation sites can be identified by exploiting any conventions that programmers may have used for naming the variables in the program. Automatic string-searching tools are used to search the source code—or alternatively, just the identifiers in a tokenized version of the source code—with respect to patterns that reflect such conventions, for example, "*date*", "*gmt*", "*yy*", etc. (where "*" is a wild-card symbol that means "match any substring").

After these techniques have been used to identify candidate sites at which dates are manipulated, this information can be "amplified", via searching and slicing [14,7,4,9] operations, to find other potential locations of problems.

3. Path Profiling and Path-Spectrum Comparison

In path profiling, a program is instrumented so that the number of times different paths of the program execute is accumulated during an execution run. Typically, the paths of interest are loop-free intraprocedural paths. The distribution of paths from an execution of the program is called a *path profile* or a *path spectrum*. We are sometimes just interested in Boolean information (which paths were executed? which were not?), but other times we are interested in the frequencies with which paths were executed. This corresponds to considering a path spectrum as either a set of paths or a multi-set of paths, respectively. The observation underlying our technique for applying program profiling to software testing is that differences between path spectra obtained from different runs of a program—with different input datasets—can be used to identify paths that are good candidates for being *data*-dependent computations.

For example, in the Y2K problem, by choosing input datasets to hold all factors constant except the way dates are used in the program, any differences in the path spectra from different execution runs can be attributed to *date*-dependent computations in the program. In particular, one would obtain path spectra from execution runs of the program in which the program is run on pre-2000 data and post-2000 data (or data that is likely to bring to light whatever "date vulnerability" we are trying to test). By comparing the two path spectra, paths along which the program performed a new sort of computation during the post-2000 run can be identified, as well as paths—and hence computations—that were no longer executed during the post-2000 run.

Our thesis is that this technique provides a good heuristic for identifying data-dependent computations. The basis for this belief is that a path spectrum provides an approximate characterization of the program's behavior, in the following sense:

> The program's execution paths serve as representatives for a set of execution states: Consider the set of all possible execution states of the form (pt, σ), where σ is a store value and pt is not an arbitrary program point, but one occurring at the beginning of a path p that the profiler is prepared to tabulate. In terms of characterizing the program's execution behavior, two execution states (pt, σ_1) and (pt, σ_2) are "similar" if they both cause the program to proceed from pt along execution path p. Path p serves as a representative of this equivalence class of similar execution states.

Differences in the path spectra obtained during two runs of a program on different inputs indicate differences in the (equivalence classes of) execution states encountered, and hence are a reflection of differences in the program's behavior due to the differences in the input. In the case of runs using pre- and post-2000 data, differences in the path spectra must therefore reflect changed behavior due to date-dependent computations.

Of course, this only holds in one direction: Not all differences in behavior due to data-dependent computations will necessarily show up as differences in the (equivalence classes of) execution states encountered.

Example. Consider the program fragment shown in Figure 1, which reads and processes data from a database of customer information. (This fragment does not contain any cycles, but might appear as part of a loop in a larger program. Path profiling in programs with loops is typically carried out by considering loop-free segments of the program. See below, or reference [2] for more discussion of this issue.)

For purposes of this example, assume that years are represented with only two digits and that no person recorded in the database who is younger than fifteen years old possesses a college

```
a: birth_year := read()
   has_college_degree := read()
   purchases := read()
   age := current_year() − birth_year
if age < 15 then
   b: . . .
else
   c: . . .
fi
if has_college_degree = true then
   d: . . .
else
   e: . . .
fi
if purchases > 3 then
   f: . . .
else
   g: . . .
fi
```

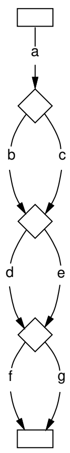


Figure 1. A program fragment that reads and processes data from a database of customer information, and its control-flow graph.

degree. Because of the latter assumption, no path from a pre-2000 run can begin with the prefix $[a,b,d]$.

Now consider a post-2000 run (*e.g.*, a simulated post-2000 run in which the system clock has been set ahead so that *current_year*() returns a value representing a year in the future, say 00, representing the year 2000), and suppose that the program reads in data about someone born in 1956 who possesses a college degree: The initialization code in region a would set *age* to $00 - 56 = -56$; because the test $-56 < 15$ evaluates to true, region b would be executed; because the person possesses a college degree, region d would be executed; finally, either region f or g would be executed. In either case, the program performs a faulty computation: The path executed is a path that should only be executed when a record is encountered for a person younger than fifteen who possesses a college degree. Because no such paths are ever executed during the pre-2000 run, the path-spectrum-comparison technique would detect the fact that the program performed a new sort of computation during the post-2000 run.

In addition, other anomalies may be detected: The pre-2000 run could very well execute paths with the prefix $[a,c]$. Because in the post-2000 run the value of *age* is always negative, the post-2000 run would never execute such paths.

The following table shows path spectra that might be accumulated during pre-2000 and post-2000 execution runs (assuming that the fragment occurs in a loop, so that it is executed multiple times):

Run	Paths Executed							
	$[a,b,d,f]$	$[a,b,d,g]$	$[a,b,e,f]$	$[a,b,e,g]$	$[a,c,d,f]$	$[a,c,d,g]$	$[a,c,e,f]$	$[a,c,e,g]$
pre-2000			●	●	●	●	●	●
post-2000	●	●	●	●				

These spectra show clearly that the pre-2000 and post-2000 behavior of the program is not the same: Paths $[a,b,d,f]$ and $[a,b,d,g]$ occur in the post-2000 run, but do not occur in the pre-2000 run; paths $[a,c,d,f]$, $[a,c,d,g]$, $[a,c,e,f]$, and $[a,c,e,g]$ occur in the pre-2000 run, but do not occur in the post-2000 run. □

Each path in a path spectrum represents a sequence of edges in the program's control-flow graph. From two path spectra, *new_spectrum* and *old_spectrum*, the path-spectrum-comparison technique reveals paths of *new_spectrum* that are not found in *old_spectrum*, and vice versa. Given a path of *new_spectrum* (resp., *old_spectrum*) that does not occur in *old_spectrum* (*new_spectrum*), we can determine the shortest prefix of the path that distinguishes it from all of the paths in *old_spectrum* (*new_spectrum*). For the Y2K problem, such path prefixes furnish a programmer with even more precise information about what contributes to the differences in behavior between the pre-2000 and post-2000 runs:

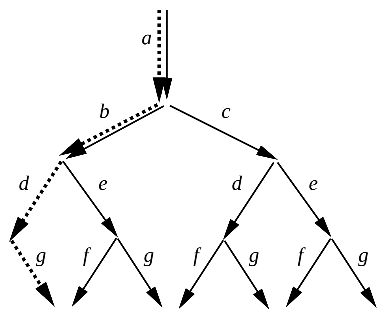
- Let p be an execution path that was executed during the post-2000 run but not during the pre-2000 run. By finding the shortest prefix of p that is not a prefix of any path executed during the pre-2000 run, we identify the critical portion of p that represents a new sort of computation (or state-transformation pattern) performed during the post-2000 run. The programmer can focus on this prefix of p to locate the date-dependent code, which very likely needs to be rewritten.
- Similarly, let q be an execution path that was executed during the pre-2000 run but not during the post-2000 run. The shortest prefix of q that is not a prefix of any path executed during the post-2000 run identifies the critical portion of q that represents a computation (state-transformation pattern) no longer performed during the post-2000 run. Again, the programmer can focus on this prefix of q to locate the date-dependent code.

Example. In the example program, paths $[a,b,d,f]$ and $[a,b,d,g]$ of the post-2000 run do not occur in the pre-2000 run. For both paths, the shortest prefix that is not a prefix of any path executed during the pre-2000 run is $[a,b,d]$. In asking the question "Why is the path $[a,b,d]$ executed during the post-2000 run?", the programmer would be led to ask the question "How can it be that *age* is less than 15 and *has_college_degree* is true?", which would in turn lead him to the statement that computes *age* as a function of *current_year* ().

Conversely, paths $[a,c,d,f]$, $[a,c,d,g]$, $[a,c,e,f]$, and $[a,c,e,g]$ of the pre-2000 run do not occur in the post-2000 run. For all of these paths, the shortest prefix that is not a prefix of any path executed during the post-2000 run is $[a,c]$. In this case, the programmer would be led to ask the question "Why is the path $[a,c]$ never executed during the post-2000 run? That is, why is the value of *age* always less than 15 during the post-2000 run?" Again, the programmer is led to the statement that computes *age* as a function of *current_year* (). □

One can find the shortest prefix of a path p that is not a prefix of any executed path in a spectrum S using a trie structure on S [12]: The first edge of p that "deviates from the trie" identifies the edge at which p veers into "unknown territory", and the prefix of p, up to and including this edge, is the shortest prefix of p that distinguishes p from S.

Example. The solid arrows in the diagram below show the trie for the pre-2000 spectrum.



The dotted edges show path $[a,b,d,g]$ (which occurs during the post-2000 run). The shortest prefix of $[a,b,d,g]$ that is not a prefix of any path executed during the pre-2000 run is $[a,b,d]$. □

The path-spectrum-comparison technique is not tied to any particular path-profiling method. Furthermore, there are a wide variety of options in how one performs the instrumentation required to gather information about what paths execute. Instrumentation can be performed at any one of a number of levels:

- At the source-code level, as a source-to-source transformation.
- As part of compilation, by extending a compiler to use its intermediate representations for the purpose of determining where to introduce instrumentation instructions.
- As an object-code-level transformation, by modifying object-code files (such as UNIX ".o" files).

- As a post-loader transformation, by modifying executable files (such as UNIX "a.out" files) [5,13,6].

One could even use different instrumentation methods on different parts of the system.

Although any method for generating path profiles could be used, it is only recently that methods have been devised for obtaining path profiles with acceptable overheads [2,1]. In particular, the Ball-Larus work relies on a clever method for numbering the paths in a program.[2] Their numbering scheme labels the edges of the program's control-flow graph with numbers such that, for every path from *Start* to *Exit*, the sum of the edge labels along the path corresponds to a *unique* number in the range [0 .. *num_paths_from* (*Start*) − 1]. That is, the following properties hold:

(1) Every path from *Start* to *Exit* corresponds to a number in the range [0 .. *num_paths_from* (*Start*) − 1].
(2) Every number in the range [0 .. *num_paths_from* (*Start*) − 1] corresponds to some path from *Start* to *Exit*.

Ball and Larus report that execution-time overheads on the order of only 30–40% can be achieved with their method for collecting path profiles [2].

Example. Returning to our running example, Figure 2 shows how the control-flow graph of the program fragment that reads and processes data from a database of customer information would be annotated. Each box is annotated with the number of paths from that node to the final node of the fragment; each edge is annotated with the number that would be assigned by the Ball-Larus numbering scheme. Note that the sum of the edge labels along each path from the beginning to the end of the graph falls in the range [0 .. 7], and that each number in the range [0 .. 7] corresponds to exactly one such path.

The instrumented version of the program's source code is shown on the left in Figure 2. Statements that increment counter *r* have been introduced so that at the end of the fragment its value indicates which path through the fragment was executed. This value is then used to increment the appropriate element of array *profile*, which maintains the frequency distribution of paths executed. (Alternatively, *profile* could maintain just a Boolean indicator of whether the path is ever executed.) □

Profiles obtained from the instrumented program can be displayed in the fashion shown below, where paths are arranged on the *x*-axis according to the path number, and the *y*-axis is used to indicate either the execution frequency or just a Boolean indicator of whether the path was executed at all. The spectra discussed earlier would be displayed as follows:

---

[2]The Ball-Larus path-numbering scheme applies to an acyclic control-flow graph with a unique source node *Start* and a sink node *Exit*. Control-flow graphs that contain cycles are modified by a preprocessing step to turn them into acyclic graphs:

> Every cycle must contain one backedge, which can be identified using depth-first search. For each backedge $w \rightarrow v$, add edges $Start \rightarrow v$ and $w \rightarrow Exit$ to the graph. Then remove all of the backedges from the graph.

The resulting graph is acyclic. In terms of the ultimate effect of this transformation on profiling, the result is that we go from having an infinite number of unbounded-length paths in the control-flow graph to having a finite number of bounded-length paths. A path *p* in the original graph that proceeds several times around a loop will, in the profile, contribute "execution counts" to several smaller acyclic paths whose concatenation makes up *p*. In particular, the paths from *Start* to *Exit* in the modified graph correspond to acyclic paths in the original graph (where following the edge $Start \rightarrow v$ that was added to the modified graph corresponds to following backedge $w \rightarrow v$ in the original graph and beginning a new path at *v*, and following the edge $w \rightarrow Exit$ that was added to the modified graph corresponds to ending the path in the original graph at *w*). (Throughout the paper, when we refer to the "control-flow graph", we mean the transformed (*i.e.*, acyclic) version of the graph.)

```
a: r := 0
    birth_year := read()
    has_college_degree := read()
    purchases := read()
    age := current_year () − birth_year
if age < 15 then
    b: . . .
else
    c: r := r + 4
        . . .
fi
if has_college_degree = true then
    d: . . .
else
    e: r := r + 2
        . . .
fi
if purchases > 3 then
    f: . . .
else
    g: r := r + 1
        . . .
fi
h: profile [r ] := profile [r ] + 1
```

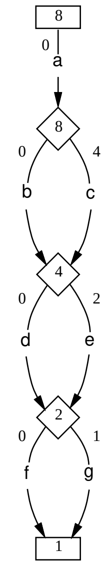
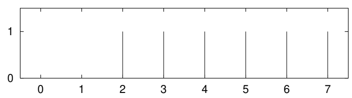
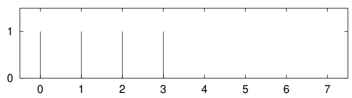


Figure 2. The instrumented version of the program fragment that reads and processes data from a database of customer information, and the program's annotated control-flow graph.



Pre-2000 spectrum



Post-2000 spectrum

4. Other Applications of Path-Spectrum Comparison in Software Testing

The path-spectrum-comparison technique is applicable to a wide range of problems that arise in software testing—certainly more than just ones that arise in the Y2K problem. Some of the ways to enlist path-spectrum comparison in the cause of providing better tools for testing software are described below.

Systems that Warn of Possible Errors Within Themselves

As described thus far, the spectra that are compared come from different runs of a program. However, the underlying principle is simply that "information about possible execution problems can be obtained by comparing two spectra". The spectra do not necessarily have to be from different runs of the program. All we care about is that there are two spectra to be compared (and that the spectra provide some sort of behavior signature). The spectra could be obtained from two or more runs (as in the application of the technique to the Y2K problem); however, there are situations in which it would be meaningful to compare spectra obtained during a single run. In particular, one way to detect unusual behavior in programs would be to make use of path-spectrum comparison to build systems that warn of possible errors within themselves. When the program detects an "oddball path", the program would signal that such a situation has just occurred—*i.e.*, to warn the user or system tester that the program has just gone down a possibly bad path. (The system could issue the warning directly to the user, to a dialog box, to the console window, or to a log file.)

Two situations in which this approach would be useful are: (i) when a system is being tested, and (ii) for building systems that warn of possible errors within themselves. In both cases, the idea is to have the system compare each path executed by the program with the paths executed so far. When a new path is discovered (*i.e.*, when the path is executed for the first time) the program would signal that a possibly erroneous computation has just occurred. Of course, one would want to wait until the program had run for a while before starting to issue such warnings, but after a break-in or warm-up period it would begin to be useful to gather such information.

Information about such oddball paths could provide important clues that would help in tracking down a bug once a symptom comes to light. For example, if a new path coincides with a program crash, then a bug report with details of the oddball path (or the last few oddball paths) could provide important information to the developers.

A variation on the idea would be to collect the spectrum of paths executed when a program is run on the regression test suite, and instrument the system to report when it executes a path that was never executed when running any of the tests. The execution of a path outside this set would be treated as an unusual occurrence (and either reported or logged). (The presence of such a path implies that the test suite used for regression testing did not contain a test for a situation that actually does arise. This information could be used as a guide for extending the test suite.)

Testing Which Parts of a System are Affected by a Modification

Another variation on path-spectrum comparison could be used to support the testing of bug fixes and other small changes to a system. The goal here would be to understand whether the only behavioral changes introduced by a modification were to the intended parts of the system. The idea is to use path-spectrum comparison as a heuristic method for understanding the magnitude of behavioral changes between two versions of a program.

In this context, the comparison that needs to be carried out is somewhat different from what has been discussed earlier: Instead of comparing spectra from two runs of the *same* program on *different* data, one would compare spectra from two runs of a (slightly) *different* program on the *same* data. As before, the premise that “states are similar if they proceed down the same path” provides the justification for why it makes sense to be comparing path spectra (even though they now come from execution runs of *different* programs).

Of course, one expects there to be differences between the two spectra obtained from the two versions of the program. For example, one would expect to see differences on the input that elicits the bug in the original program. The purpose of comparing the path spectra would be to obtain information about the extent of actual changes in behavior. One wants to make sure that a small change in the program text does not lead to radical changes in the behavior. The behavior of most of the unmodified parts of the system should be unaffected by a modification. The programmer can use the information obtained from path-spectrum comparison to develop an understanding of the actual magnitude of behavioral differences that a bug fix introduces.

In order to carry out comparisons between paths from two different programs, a concordance between paths in the old program and paths in the new program would be needed. The instrumentation strategy used affects how difficult it is to provide such a concordance: It would not be too hard to establish a correspondence between paths in the old and new programs when source-code instrumentation is used, but it would be much more difficult to do so when instrumentation is carried out on object-code files or executable files.

Testing for Inconsistent Data

Another potential application of path-spectrum comparison is to the “data-hygiene” problem. The goal here is to identify data in a database or file that is contaminated, or inconsistent with the assumptions about the data that the program relies on. Our hypothesis is that some contaminated data items will cause the program to take unusual paths through the code (but ones that do not actually crash the system). Presumably the percentage of contaminated data is low; thus, the idea behind using path-spectrum comparison is to use information about infrequently executed paths to identify possibly contaminated data in the database. Any peculiar paths (*i.e.*, paths with count 1 or low relative frequency in the path spectrum) when the program is run against the database would be taken as a signal that the program was processing possibly contaminated data. To actually identify the contaminated data, one would need the instrumented program to gather some additional information in order to link the low-frequency paths back to the inputs that were most recently read in at the times the path was executed.

5. Implementation and Preliminary Results

A prototype tool for gathering and comparing path spectra, called DYNADIFF, has been built at the University of Wisconsin. DYNADIFF runs under Solaris on Sun SPARCstations. It uses Tcl/Tk to implement a graphical user interface, and Larus's implementation of the Ball-Larus path-profiling algorithm as the underlying machinery for generating path spectra. The path profiler instruments executable files, so programs can be written in any language (as long as the compiler for the language obeys certain calling conventions) or even in a mixture of languages.

The goal of DYNADIFF's user interface is to allow one to collect up, and perform difference operations on, collections of path profiles. (The DYNADIFF user can display path profiles as spectra of the kind shown earlier: At present, the system treats each path profile as merely a set of paths; that is, the frequency counts of the number of times each path executed is ignored, and an executed path in a spectrum is displayed as a stick of height 1.) Spectra have links back to the source code: Clicking on the stick that represents a path brings up an *emacs* window with the elements of the path displayed in a special color.

DYNADIFF is organized around the notions of *profiles* and *workspaces*: Collections of profiles can be selected and placed in named workspaces. Because we are interested in path-spectrum differences, when path profiles from a workspace are displayed as spectra, each spectrum shows only paths that were executed in at least one of the profiles of the workspace but not in all of the profiles.

As part of calling up spectrum differences, the user forms sub-partitions of the profiles in a workspace. The profiles in a workspace are partitioned into three groups, which we will call *A*, *B*, and *Other*. (That is, *A*, *B*, and *Other* are each sets of profiles.) Spectrum differences are displayed by showing path sticks for paths that are executed by all profiles in *A*, but not by some profile in *B*, and vice versa. Clicking on one of the path sticks brings up an *emacs* window with the statements of the last edge of the shortest distinguishing prefix of the path displayed in one special color, the rest of the shortest distinguishing prefix displayed in a second special color, and the rest of the elements of the path displayed in a third special color.

DYNADIFF has been used in several small experiments to test the efficacy of the path-spectrum-comparison technique. One experiment that we carried out with DYNADIFF was aimed at testing the ability of path-spectrum comparison to identify bugs in leap-year calculations. This experiment involved the UNIX *cal* utility, which, given a month and a year as input, prints the calendar for that month. The *cal* program does not actually have a leap-year bug: It calculates correctly that the year 2000 is a leap year. However, because our goal was merely to determine whether path-spectrum comparison would be able to identify leap-year calculations, this did not matter—we tested the method's sensitivity to leap-year calculations by comparing spectra from leap years and non-leap years. Path spectra obtained from runs that we expected would involve leap-year calculations (*e.g.*, from inputs like "*cal 2 1992*", "*cal 2 1996*", etc.) were compared against spectra obtained from runs that we expected not to involve leap-year calculations (*e.g.*, "*cal 2 1997*", "*cal 2 1998*", etc.).

For example, in a trial with workspace-partition *A* consisting of the profile from a run with input "*cal 2 1992*" and *B* consisting of the profile from a run with input "*cal 2 1997*", there was

- One path that was executed during the run with input "*cal 2 1992*", but not during the run with input "*cal 2 1997*".
- One path that was executed during the run with input "*cal 2 1997*", but not during the run with input "*cal 2 1992*".

Figure 3 shows the path that was executed during the run with input "*cal 2 1992*", but not during the run with input "*cal 2 1997*", as well as the shortest prefix of the path that distinguishes it from all paths of the "*cal 2 1997*" run. To understand the code shown in Figure 3, it helps to know that the routine "jan1" receives a year value as its parameter, and returns a number in the range [0 .. 6] that represents the day of the week on which January 1 falls that year. The values 0 through 6 correspond to Sunday through Saturday, respectively. The switch statement chooses one of three cases, depending on the difference (in terms of number of days of the week) between jan1(y) and jan1(y+1). The switch value is 1 in the case of an ordinary, non-leap year; 2 in the case of a leap year; and 5, represented by the default case, in 1752, the year that England and the Colonies shifted from the Julian to the Gregorian calendar. The default case is used to make a minor adjustment to one of the program's internal tables, which has an effect elsewhere on how the calendar for September 1752 is created.

Figure 3 also illustrates a small glitch due to the fact that the path profiler we used instruments executable files. The program shown in Figure 3 has an additional statement, "foo = foo + 1;" that we added in "case 2" of the switch statement. With the original program,

```
cal(m, y, p, w)
char *p;
{
    register d, i;
    register char *s;
    int foo = 0;

    s = p;
    d = jan1(y);
    mon[2] = 29;
    mon[9] = 30;
    switch((jan1(y+1)+7-d)%7) {
        case 1:    /*   non-leap year   */
            mon[2] = 28;
            break;
        default:   /*   1752   */
            mon[9] = 19;
            break;
        case 2:    /*   leap year   */
            foo = foo + 1;   /* Statement added so that something in the leap-year case */
            break;           /* could be highlighted */
    }
    for(i=1; i<m; i++)
        d += mon[i];
    d %= 7;
    s += 3*d;
    . . .
```

Figure 3. The code displayed in ***Times-BoldItalic***, **Helvetica-Bold**, and **Times-Bold** indicates a path that was executed during a run with input "*cal 2 1992*", but not during a run with input "*cal 2 1997*". The code shown in ***Times-BoldItalic*** and **Helvetica-Bold** indicates the shortest prefix of the path that distinguishes it from all paths of the "*cal 2 1997*" run. The code shown in **Helvetica-Bold** indicates the last edge of the shortest distinguishing prefix (*i.e.*, **switch((jan1(y+1)+7-d)%7)** → **foo = foo + 1;**).

in which "case 2" was empty, we were initially confused by the path that DYNADIFF highlighted. No part of "case 2" was highlighted, and we did not at first recognize that the path actually did go into that branch of the switch statement. The reason for this was that the current version of DYNADIFF uses information generated by the compiler to map from addresses in executable files to lines in the source code. Our confusion was caused by the fact that the compiler had not generated any instructions for the empty case, and so DYNADIFF did not have the information it needed to highlight "case 2". In Figure 3, the statement "foo = foo + 1;" was added so that something existed in the body of "case 2" that could be highlighted. (If DYNADIFF were to perform path profiling via source-code instrumentation, it would not have this problem.)

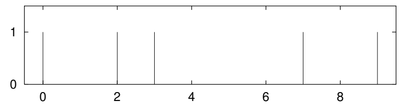
Other experiments that we carried out with DYNADIFF were aimed at testing the ability of path-spectrum comparison to identify date-rollover problems. Although most UNIX programs do not have a Y2K problem, many have a "Year 2038 problem":

> The UNIX *time* function, which reports the number of seconds since January 1, 1970, rolls over from 2**31–1 (*i.e.*, 01111111111111111111111111111111) to 2**31 (*i.e.*, 10000000000000000000000000000000)—and thus turns negative—on Tuesday, January 19, 2038 at 03:14:08 UTC.

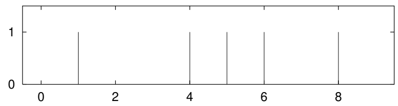
Thus, we can demonstrate the *principle* of using path-spectrum comparison for diagnosing possible Y2K problems by applying DYNADIFF to compare spectra generated from normal runs against spectra generated from runs in which the result of *time* has been "time-warped" into the future. For example, the UNIX *cal* utility (when called with no arguments) prints the calendar for the current month. The "current month" is obtained by calling *time*. Just after *time* overflows, *cal* erroneously prints the calendar for December 1901.

Below is an example of the spectral differences that DYNADIFF displays (for a version of *cal* modified to optionally permit "time-warping") when it compares a spectrum from a run from

February 1997 against a run made with *time* set to emulate a run in February 2039 (*i.e.*, post-rollover):



February 1997 spectrum



February 2039 spectrum

We see that the path-spectrum comparison technique does indeed detect that the two runs of the program had different behaviors. (Similar results were obtained in two other experiments that we carried out on the *rcs* and *ncftp* utilities.)

One of the paths executed in the year-2039 run—but not in the year-1997 run—is shown in Figure 4. ***Times-BoldItalic*** indicates the longest prefix that the path shares with some path in the year-1997 path set; **Helvetica-Bold** indicates the endpoints of the first edge that distinguishes the path from all of the paths in the year-1997 path set; **Times-Bold** indicates the remainder of the path.

The typefaces shown in Figure 4 indicate that the year-2039 run executed the loop

```
while (rem < 0) {
    rem += SECSPERDAY;
    --days;
}
```

at least once, whereas the year-1997 run did not execute this loop at all. This observation, coupled with an examination of the statements that set the value of variable rem immediately before the loop executes, allows us to deduce that the value of *timep must be negative on

```
static void
timesub(timep, offset, tmp)
const time_t * const                     timep;
const long                               offset;
register struct tm * const               tmp;
{
    register long                        days;
    register long                        rem;
    register int                         y;
    register int                         yleap;
    register const int *                 ip;

    days = *timep / SECSPERDAY;
    rem = *timep % SECSPERDAY;
    rem += offset;
    while (rem < 0) {
        rem += SECSPERDAY;
        --days;
    }
    while (rem >= SECSPERDAY) {
        rem -= SECSPERDAY;
        ++days;
    }
    . . .
```

Figure 4. The code displayed in ***Times-BoldItalic***, **Helvetica-Bold**, and **Times-Bold** indicates a path that was executed during a year-2039 run but not during a year-1997 run. The code shown in ***Times-BoldItalic*** and **Helvetica-Bold** indicates the shortest prefix of the path that distinguishes it from all paths of the year-1997 run. The code shown in **Helvetica-Bold** indicates the last edge of the shortest distinguishing prefix (*i.e.*, **(rem < 0)** → **rem += SECSPERDAY;**).

entry to procedure timesub. The program renovator could then use this information (for example, by employing program slicing [14,7,4,9]) to trace back to the source of the problem.

6. Final Remarks

In several places in the paper, we have referred to a path spectrum as a "behavior signature for a run of the program", and have advanced the general principle that "information about differences in execution behavior can be obtained by comparing two behavior signatures". One has to be a bit careful about the notion of a behavior signature, however. For instance, a trace of the program counter could also be considered to be a behavior signature for a run of the program.

This leads to the conclusion that not all behavior signatures are equally valuable. Consider again the Y2K problem: The comparison of two traces of the program counter, generated during separate runs of the program, would yield only a small amount of useful information that could be used by a programmer trying to determine where the program performs problematic date manipulations. For example, the first point at which the traces diverge could be attributed to a date-dependent computation, but extracting additional information about other problematic date manipulations in the program would be difficult. However, when the two traces are transformed into path spectra, they take on a form that does provide utility for this task. This suggests the following analogy:

> In physics, it is often more appropriate to work with the Fourier transform of a function rather than with the function itself. The reason is that the Fourier transform reveals the fundamental excitation modes of a system. Manipulations of Fourier transforms operate directly in terms of these fundamental modes.
>
> Similarly, we have identified situations in which it is more appropriate to work with path spectra rather than with traces of the program counter. The reason is that a path spectrum reveals the "fundamental modes" of the program that were "excited" during an execution run. Manipulations of path spectra operate directly in terms of the program's fundamental excitation modes.

(The fact that a Fourier transform is information preserving, whereas a path profile loses information available in an execution trace, does not invalidate the analogy. The loss of information is tied up with a different issue, namely that of abstraction. That is, the creation of a path spectrum involves both a transformation that reveals underlying structure, as well as an abstraction of information available in the original trace.)

One reason for casting things in these terms is that it suggests an avenue for further research: Are there other "transforms" to be discovered that would tease out interesting information (similar to our "fundamental modes of a program excited during an execution run") from other kinds of data about programs, execution runs, etc.?

References

1. Bala, V., "Low overhead path profiling," Tech. Rep., Hewlett-Packard Labs (1996).
2. Ball, T. and Larus, J., "Efficient path profiling," in *Proc. of MICRO-29*, (Dec. 1996).
3. Clarke, L.A., Podgurski, A., Richardson, D.J., and Zeil, S.J., "A comparison of data flow path selection criteria," pp. 244-251 in *Proc. of the Eighth Int. Conf. on Softw. Eng.*, IEEE Comp. Soc. Press, Wash., DC (1985).
4. Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.* **12**(1) pp. 26-60 (Jan. 1990).
5. Johnson, S.C., "Postloading for fun and profit," pp. 325-330 in *Proc. of the Winter 1990 USENIX Conf.*, (Jan. 1990).
6. Larus, J.R. and Schnarr, E., "EEL: Machine-independent executable editing," *Proc. of the ACM SIGPLAN 95 Conf. on Programming Language Design and Implementation*, (La Jolla, CA, June 18-21, 1995), *ACM SIGPLAN Notices* **30**(6) pp. 291-300 (June 1995).
7. Ottenstein, K.J. and Ottenstein, L.M., "The program dependence graph in a software development environment," *Proc. of the ACM SIGSOFT/SIGPLAN Softw. Eng. Symp. on Practical Software Development Environments*, (Pittsburgh, PA, Apr. 23-25, 1984), *ACM SIGPLAN Notices* **19**(5) pp. 177-184 (May 1984).

8. Rapps, S. and Weyuker, E.J., “Selecting software test data using data flow information,” *IEEE Trans. on Softw. Eng.* **SE-11**(4) pp. 367-375 (Apr. 1985).
9. Reps, T., Horwitz, S., Sagiv, M., and Rosay, G., “Speeding up slicing,” *SIGSOFT 94: Proc. of the Second ACM SIGSOFT Symp. on the Found. of Softw. Eng.,* (New Orleans, LA, Dec. 7-9, 1994), *ACM SIGSOFT Softw. Eng. Notes* **19**(5) pp. 11-20 (Dec. 1994).
10. Reps, T., Ball, T., Das, M., and Larus, J., “The use of program profiling for software maintenance with applications to the Year 2000 Problem,” in *Proc. of ESEC/FSE ′97: Sixth European Softw. Eng. Conf. and Fifth ACM SIGSOFT Symp. on the Found. of Softw. Eng.,* (Zurich, Switzerland, Sept. 22-25, 1997), *Lec. Notes in Comp. Sci.*, Springer-Verlag, New York, NY (1997).
11. Roper, M., *Software Testing,* McGraw-Hill, New York, NY (1994).
12. Sedgewick, R., *Algorithms,* Addison-Wesley, Reading, MA (1983).
13. Srivastava, A. and Eustace, A., “ATOM: A system for building customized program analysis tools,” *Proc. of the ACM SIGPLAN 94 Conf. on Programming Language Design and Implementation,* (Orlando, FL, June 22-24, 1994), *ACM SIGPLAN Notices* **29**(6) pp. 196-205 (June 1994).
14. Weiser, M., “Program slicing,” *IEEE Trans. on Softw. Eng.* **SE-10**(4) pp. 352-357 (July 1984).
15. Woodward, M.R., Hedley, D., and Hennell, M.A., “Experience with path analysis and testing of programs,” *IEEE Trans. on Softw. Eng.* **SE-6**(3) pp. 278-286 (May 1980).