

Towards Differential Program Analysis

Joel Winstead and David Evans
Department of Computer Science
University of Virginia
{jwinstead,evans}@cs.virginia.edu

Abstract

Differential Program Analysis is the task of analyzing two related programs to determine the behavioral difference between them. One goal is to find an input for which the two programs will produce different outputs, thus illustrating the behavioral difference between the two programs. Because the general problem is undecidable, an unsound or incomplete analysis is necessary. A combination of static and dynamic techniques may be able to produce useful results for typical programs, by conducting a search for differentiating inputs guided by heuristics. This paper defines the problem, describing what would be necessary for this kind of analysis, and presents preliminary results illustrating the potential of this technique.

1. Introduction

Notkin has argued that the future of program analysis lies in analyzing multiple versions of the same program together [4]. This allows us to amortize the cost of analysis across the development cycle, as well as to direct analysis efforts towards differences, and may allow kinds of analysis that would otherwise be intractable. We agree that this is a good strategy, and further argue that analyzing two versions of a program to find a behavioral difference is an important problem not just because it can reduce the cost of analysis, but because finding behavioral differences is a useful goal in itself: it can aid in understanding and maintaining a program as well as in recognizing unintended side effects of modifications.

When making a change to a program, either to correct a known error or to add a new feature, the consequences of the change are not always fully understood. The change may have unintended side effects that were not anticipated by the programmer, or may fail to accomplish the intended goal. The change may even have no effect at all. In order to prevent unintended side effects and verify that changes have the intended effect, it would be helpful to have an automated

analysis showing the actual effect of the modification on the program's behavior.

Programs are frequently maintained by people who are far removed from the original development process. The intended purpose of modifications in the program's history is not always clear or documented. The actual effect on the program's behavior of the presence of a particular part of the program may be unknown; a particular line may be crucial or it may have no effect at all. An analysis that shows the difference in behavior caused by the presence or absence of a particular element would assist maintainers in understanding the program.

Testing and dynamic analysis of programs could also benefit from this sort of analysis. When a change to a program is made, it is important that it is well tested. New tests may need to be added to the regression test suite to test the modification adequately. Many dynamic analysis tools depend on the quality of the test suite for a program, and may produce incorrect results if no tests exist that exercise a particular modification.

What is needed is a set of automated techniques to analyze the effect of modifications. We use *differential program analysis* as a general term to describe analyses that focus on the differences between two similar programs. In the sections that follow, we outline what such an analysis needs to do, propose some heuristics and techniques that can be used to do this analysis, and present preliminary results showing the promise of this technique.

2. Problem Definition

One goal of differential program analysis is to generate a test case that demonstrates the difference in behavior between the two programs. We assume that the behavioral difference is small relative to the input space (i.e., the two programs produce identical output for nearly all inputs). While it would be interesting to analyze changes that affect the result of every input to the program, this would require a different kind of analysis. Our goal is to find behavioral differences, not to analyze known ones. Once a difference is

found, existing techniques such as Zeller’s Delta Debugging method [9] can be used to analyze the difference.

We concentrate on analyzing two versions of the same program. The structural difference between the two programs must be small relative to the size of the program: only a few lines of code or a few procedures in the program should be different. We would like to develop techniques that take advantage of the similarities between the two programs, rather than use existing techniques to analyze the programs independently and compare the results.

Because our goals include finding unanticipated side-effects of changes, we cannot assume that an existing regression test suite is able to find all interesting behavioral differences. Regression testing finds differences in behavior that were anticipated by the designers (or testers) and specifically checked. While regression test selection [1] is a useful technique for reducing the cost of testing, it cannot reveal new differences that are not already tested by the suite. We also would like to be able to analyze undocumented programs that may not have test suites.

We assume we have a generator capable of producing a differentiating test case, but that it is not reasonable to do an exhaustive search of the input space. It is not necessary for all generated inputs to be valid; the search will eliminate inputs that both programs consider to be errors. If the difference in behavior is small relative to the input space, and we have a generator that can produce the right inputs, the analysis problem becomes one of performing a directed search to find inputs which reveal behavioral differences.

3. Approach

This kind of analysis requires solving several subproblems: we must find inputs that reach the syntactic difference, generate differences in state between the two programs, and propagate these differences to the output.

The two programs will always produce the same output for a given input unless, at some point, they execute different instructions. Therefore, in order to find test cases that result in different output, we must first figure out how to reach the syntactic changes to the program. This subproblem is itself undecidable, but incomplete solutions have been proposed for it using search techniques such as simulated annealing [6] or genetic algorithms [3] [5]; these techniques use fitness functions to generate test inputs that reach particular parts of a program. Symbolic execution and constraint solving is also a possible approach.

Once the syntactic changes in the program have been reached, it is also necessary for a difference in state to result, and for this difference to be propagated through the programs far enough to result in different output. It is possible for a modification to produce changes in intermediate values without producing any difference in the end result.

Narrowing the input space to inputs that reach the modification will not always be sufficient: it will still be necessary to search this space to find inputs that result in actual differences in output.

Tracey et al. [6] show how to use simulated annealing to evolve test inputs that cause a program to reach a specified point in the code. This is accomplished by determining what branches the program must take to reach the point of interest, and developing a fitness function that evaluates how close the program is to taking the correct branches. For branches that the program should take, the condition for the branch is transformed into an expression that measures how close the program is to taking the branch, and these expressions are combined to form a fitness function which is used to evolve test inputs that cause the program to reach the desired point. In order to apply this technique to finding differences between programs, new fitness functions must be constructed that compare the behavior of the two programs and guide the search towards input that is likely to reveal differences.

In order to direct the search towards an input that produces an actual behavioral difference, we must have some way to measure how close a particular input is to achieving this goal even if the goal has not yet been reached. While not all inputs will produce a difference in output, some inputs may produce different intermediate values, or other measurable differences in execution that may be important cues for finding inputs that produce a behavioral difference. We propose several heuristics that may be useful for guiding a search towards inputs that produce actual differences in output.

First, we note that boundary conditions in the two programs indicate what decisions the programs are making, and that differences in behavior often lie along boundary conditions. Selecting test cases that exercise boundary conditions in the two programs is a promising way to find differences. Focusing attention on boundaries that exist in one program but not the other is particularly interesting, because these decisions lead to paths that are not in both programs, and reaching these paths may reveal different behavior. This last heuristic takes advantage of known similarities between the programs to focus on the difference, and has the potential to be more useful than approaches that analyze the programs separately.

If we evolve test sets, instead of evolving single test cases, there are additional heuristics we can use. We can select for test sets that maximize the total number of paths executed, or other coverage metrics. We could modify these coverage metrics to include only those paths that reach the syntactic difference between the programs, which also takes advantage of the similarities between the programs.

Evolving test sets also allows us to compare the ways the two programs map the input space into paths through

the program. If program P_1 maps two inputs I_1 and I_2 to the same path, but program P_2 maps inputs I_1 and I_2 to two different paths, this reveals something about how the programs divide the input space, even if the output is the same. Selecting for test sets that do not produce isomorphic mappings from inputs to paths in this way may lead to revealing behavioral differences because they indicate regions of the input space where the two programs do not handle the input in the same way.

In addition to these heuristics, ongoing work by Xie and Notkin [8] examines comparing program spectra combined with various heuristics to identify possible faults in modified programs even in cases where no actual differences in output result. Program spectra are signatures of program behavior (such as the distribution of paths taken, procedures executed, and values modified by the program) that can be used to characterize the program's execution. Harrold et al. [2] also investigate the effectiveness of various program spectra in identifying differences between programs. These studies focus on identifying differences in execution that occur in regression tests of a program, not on guiding a search for inputs that produce actual behavioral differences. However, some of the heuristics identified there may be useful in constructing fitness functions that are useful for this purpose.

If no differentiating test case can be found, this does not necessarily mean that no behavioral difference exists. It would be useful to have some way of measuring how thorough a search has been conducted, because this would provide a measure of confidence that the two programs actually have the same behavior. This could be done by estimating how much of the relevant search space has been tested, or by coverage of the modified parts of the programs. Mutation analysis [7] has been used to evaluate the effectiveness of a test suite by how well it can identify differences between the original and modified programs. It may be necessary to develop new coverage metrics that take into account the special problem of covering differences in code.

4. Preliminary Results

We have developed a small system to explore some of the techniques described above. The system uses less than 1,000 lines of Java code, and is capable of evolving test cases that show behavioral differences in small programs. The programs must be instrumented by hand to compute the fitness functions, but this could be automated in the future. We will illustrate the system using a simple example.

The short procedure in Figure 1 classifies a triangle by comparing the lengths of its three sides: it returns a value indicating whether or not the three lengths given can form a triangle, and if so, whether the triangle is equilateral, isosceles, or scalene, and whether the largest angle is right, obtuse,

```
int classify( int a, int b, int c) {
    int kind = UNKNOWN;
    if (a + b <= c || b + c <= a || c + a <= b)
        return INVALID_TRIANGLE;

    if ( a*a + b*b == c*c || b*b + c*c == a*a
        || c*c + a*a == b*b)
        kind |= RIGHT_TRIANGLE;
    else if ( a*a + b*b > c*c
            && a*b + c*c > a*a
            && c*c + a*a > b*b)
        kind |= ACUTE_TRIANGLE;
    else
        kind |= OBTUSE_TRIANGLE;

    if (a==b || b==c || c==a)
        if (a==b && b==c)
            kind |= EQUILATERAL_TRIANGLE;
        else
            kind |= ISOSCELES_TRIANGLE;
    else
        kind |= SCALENE_TRIANGLE;

    return kind;
}
```

Figure 1. Triangle classification procedure

or acute. A different version of this procedure (not shown) lacks the `|| c==a` (shown in the box).

This procedure is simple enough that we can easily see the effect of the modification by inspecting it: it will, in some cases, incorrectly classify an isosceles triangle as scalene (for example, the triangle with sides (3,4,3) would be classified as scalene, even though sides a and c are the same length). However, we will use it as an example to demonstrate how an automated tool could generate test cases that demonstrate the difference using the boundary condition heuristic.

The idea behind the heuristic is that test executions that are at or near boundary conditions in the program are more likely to reveal differences. We can develop such test cases by instrumenting the program to compute a measure of nearness to boundaries, and use this measure to guide the search for differentiating test cases. At each decision point in the program, the conditional expression is converted into an expression measuring how far the program was from making the opposite decision, similar to the technique Tracey et al. used to select inputs that reach a particular point in the program [6]. We use the minimum of all of these values to measure how close the program was to taking a different path for a particular test case. We can then use this fitness function to guide a genetic algorithm to

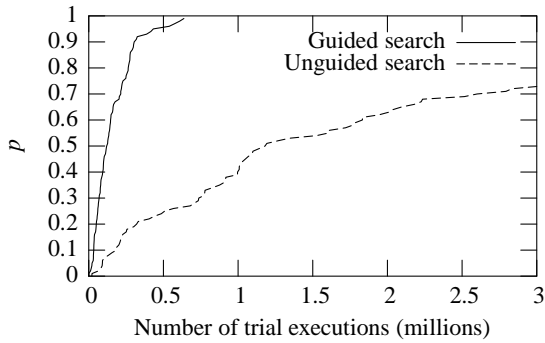


Figure 2. Probability of finding difference vs. number of trial executions

evolve test cases that cause the program to reach boundary conditions.

Preliminary results show that this technique works for the triangle classification example given above: the guided search was able to find a differentiating test case using significantly fewer test executions than a random, unguided search. The graph in Figure 2 shows that fifty percent of the time, the guided search was able to identify the difference in 120,000 trial executions or fewer, while the unguided search required over a million trial executions before having a 50% probability of finding one.

This technique works particularly well for this (admittedly contrived) example because the behavioral difference lies along one of the boundary conditions: $c=a$. For programs that have more complicated control flow, and a more complicated relationship between the input and the control flow, this is not sufficient. However, there are several ways this technique can be improved. We are currently exploring focusing on decisions that are made in only one program but not in the other in order to guide the search, rather than looking at all boundary conditions. We are also examining the use of static analysis to determine which decisions are most important and which decisions must be made to reach the changed portion of the program.

5. Summary

The modern development process, using version control systems like CVS and online repositories such as SourceForge, makes available many related versions of the same programs. We should take advantage of this opportunity to analyze related versions of programs to better understand them. It is important to develop techniques to analyze two versions of the same program together, not only because it could reduce the overall cost of testing and analysis, but because it could reveal important facts about the differences

between the programs. This kind of information would be useful in avoiding unintended side effects, understanding the development history of undocumented programs, and in identifying the actual effects of particular parts of the program.

Many of the hard problems that must be solved before we can achieve the goals of differential program analysis have undergone rapid progress recently, such as using global search techniques to evolve inputs that reach particular parts of programs [6], and using program spectra to compare related versions of the the same program [2] [8]. We are optimistic that we are near the point when techniques can be combined in ways that enable useful and revealing analyses of the differences between two similar programs.

References

- [1] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*, 2001.
- [2] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194, 2000.
- [3] C. C. Michael, G. E. McGraw, M. A. Schatz, and C. C. Walton. Genetic algorithms for dynamic test data generation. Technical Report RSTR-003-97-11, RST Corporation, Sterling, VA, May 1997.
- [4] D. Notkin. Keynote: Longitudinal program analysis. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE 2002)*, page 1, Charleston, SC, November 2002.
- [5] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *Journal of Software Testing, Verification, and Reliability*, 9(4):263–282, 1999.
- [6] N. Tracey, J. Clark, and K. Mander. The way forward for unifying dynamic test-case generation: The optimisation-based approach. In *International Workshop on Dependable Computing and Its Applications (DCIA)*, pages 169–180. IFIP, January 1998.
- [7] J. M. Voas and K. W. Miller. Software testability: The new verification. *IEEE Software*, 12(3):17–28, May 1995.
- [8] T. Xie and D. Notkin. Checking inside the black box: Regression fault exposure and localization based on value spectra differences. FSE Poster Session, November 2002.
- [9] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.