Guided Path Exploration for Regression Test Generation

Kunal Taneja[1], Tao Xie[1], Nikolai Tillmann[2], Jonathan de Halleux[2], Wolfram Schulte[2]
[1]Department of Computer Science, North Carolina State University, Raleigh, NC, 27695, USA
[2]Microsoft Research, One Microsoft Way, Redmond, WA, 98074, USA
[1]{ktaneja, txie}@ncsu.edu, [2]{nikolait, jhalleux, schulte}@microsoft.com

Abstract

Regression test generation aims at generating a test suite that can detect behavioral differences between the original and the modified versions of a program. Regression test generation can be automated by using Dynamic Symbolic Execution (DSE), a state-of-the-art test generation technique, to generate a test suite achieving high structural coverage. DSE explores paths in the program to achieve high structural coverage, and exploration of all these paths can often be expensive. However, if our aim is to detect behavioral differences between two versions of a program, we do not need to explore all paths in the program as not all these paths are relevant for detecting behavioral differences. In this paper, we propose a guided path exploration approach that avoids exploring irrelevant paths and gives priority to more promising paths (in terms of detecting behavioral differences) such that behavioral differences are more likely to be detected earlier in path exploration. Preliminary results show that our approach requires about 12.9% fewer runs on average (maximum 25%) to cause the execution of a changed statement and 11.8% fewer runs on average (maximum 31.2%) to cause program-state differences after its execution than the search strategies without guidance.

1 Introduction

Regression test generation aims at generating a test suite that can detect behavioral differences between the original and the new versions of a program. A behavioral difference between two versions of a program can be reflected by the difference between the observable outputs produced by the execution of the same test (referred to as a difference-exposing test) on the two versions. Developers can inspect these behavioral differences to determine whether they are intended or unintended (i.e., regression faults).

Regression test generation can be automated by using Dynamic Symbolic Execution (DSE) [6], a state-of-the-art test generation technique, to generate a test suite achieving high structural coverage. DSE explores paths in a program to achieve a high structural coverage, and exploration of all these paths can often be expensive. However, if our aim is to detect behavioral differences between two versions of a program, we do not need to explore all paths in the program as not all these paths are relevant for detecting behavioral differences.

To formally investigate irrelevant paths for exposing behavioral differences, we adopt the Propagation, Infection, and Execution (PIE) model [11] of error propagation. According to the PIE model, a fault can be detected by a test if a faulty statement is executed (E), the execution of the faulty statement infects the state (I), and the infected state (i.e., error) propagates to an observable output (P). A change in the new version of a program can be treated as a fault and then the PIE model is applicable for effect propagation of the change. Many paths in a program often cannot help in satisfying any of the conditions P, I, or E of the PIE model.

In this paper, we present an approach that uses DSE to detect behavioral differences based on the notion of the PIE model. Our approach first determines all the branches (in the program under test) that cannot help in achieving any of the conditions P, I, or E of the PIE model in terms of the changes in the program. To make test generation efficient, we develop a new search strategy for DSE to avoid exploring these irrelevant branches (including an irrelevant branch in a path leads to an irrelevant path). In addition, our approach prioritizes the flipping of branching nodes[1] in such a manner that behavioral differences are more likely to be detected earlier in path exploration.

This paper makes the following major contributions:
Approach. We propose an approach that uses DSE for efficient generation of regression unit tests. To the best of our knowledge, ours is the first approach that guides path exploration specifically for regression test generation.
Preliminary Evaluation. We have conducted experiments on an original and its 11 different new versions of a class. Preliminary results show that our approach requires about

[1]A branching node in the execution tree of a program is an instance of a conditional statement in the source code. A branching node consists of two sides: the true branch and the false branch. Flipping a branching node is flipping the execution of the program from the true (or false) branch of the branching node to the false (or true) branch.

```
     static public boolean testMe(int x, int[]  y){
1        int j=1;
2        if(x==90){
3            for(int i=0; i< y.Length; i++){
4                if(y[i] == 15)
5                    x++;
6                else if(y[i] == 16)
7                    j=2;
8                else if(y[i] == 25)
9                    return false;
10               if(x == 110)
11                   x = j+2; //x = 2*j+1
12               if(x>110)
13                   return false;
14           }
15       }
26       return false;
     }
```

Figure 1. An example program

12.9% fewer runs on average (maximum 25%) to cause the execution of a changed statement and 11.8% fewer runs on average (maximum 31.2%) to cause program-state differences after its execution than the default search startegy in Pex [10] (an automated structural testing tool for .NET developed at Microsoft Research).

2 Approach

Our approach takes two versions of the program under test as input and generates tests for these two versions. The generated tests on execution detect behavioral differences between the two versions. Our approach consists of two phases. In the first phase, we instrument the program code for regression testing. In the second phase, we perform DSE on the instrumented code using Pex [10]. We next discuss in detail the two phases of our approach.

2.1 Instrumentation for Regression Testing

Our approach first transforms the two versions of the program code such that the transformed program code is amenable to regression testing. In particular, our approach instruments both versions of the program under test. The instrumentation allows us to compare the internal behavior of running the same generated test on the two versions.

Consider the example in Figure 1. Suppose that the statement at Line 11 of `testMe` has been modified resulting in the one shown in the comment at Line 11. Figure 2 shows the new version of `testMe` after instrumentation. We insert a statement (Line 12 in Figure 2) just after any changed statement (Line 11 in Figure 1). The instrumented statement allows us to store the current value of `x` in a particular run (i.e., an explored path) of DSE. In particular, this statement results in an assertion in the generated test. The generated test can be executed on the original version of `testMe` to compare program states after the execution of the changed statement with the ones captured in the execution of the new

```
     public boolean testMe(int x, int[]  y){
          ...
10        if(x == 110){
11           x = 2*j+1;
12           PexStore.ValueForValidation("uniqueName", x);
13        }
          ...
  }
```

Figure 2. Instrumented example program after instrumentation

version. If there are multiple changed statements in the program, our approach first finds multiple regions each of which contains nearby changed statements in the program. We refer to each of such regions as a changed region in the rest of the paper. Our approach finds all the variables and fields that have been defined in a changed region and inserts statements (such as the statement at Line 12 of Figure 2) to log the value of each defined variable or field in the changed region. If a defined variable is a non-primitive type, the statement enables to compare the object graphs reachable from the logged values to compare program states. We then perform DSE on the instrumented new version of the program, as described in Section 2.2.

In our approach, we perform DSE on the instrumented new version of the program. After each run of DSE, we execute the generated test on the instrumented original version to check whether the program state is infected after the execution of a changed region.

2.2 Dynamic Test Generation

In the second phase, our approach uses Dynamic Symbolic Execution (DSE) [6] to generate regression tests for the two given versions of a program. DSE iteratively generates test inputs to cover various feasible paths in the program under test (the new version). In particular, DSE flips some branching node from a previous execution to generate a test input for covering a new path. The node to be flipped is decided by a search strategy such as depth-first search. The exploration is quite expensive since there are an exponential number of paths with respect to the number of branches in a program. However, the execution of many branches often cannot help in detecting behavioral differences. In other words, covering these branches does not help in satisfying any of the condition P, I, or E in the PIE model described earlier. Therefore, we do not flip such branching nodes in our new search strategy for finding test inputs that detect behavioral differences between the two given versions of a program.

2.2.1 Paths being Pruned

We next describe the three categories of paths that our approach avoids exploring.

Rationale E: Paths not leading to any changed region. Paths that cannot reach any changed region (denoted as δ) need not be explored. For example, consider the `testMe`

program in Figure 1. The changed statement is at Line 11 (δ). While searching for a path to cover δ, we do not need explore paths containing the `true` branch of the condition at Line 8.

Rationale I: Paths not causing any state infection. Suppose that we cover δ at Line 11 in Figure 1 using inputs `x=90` and the array `y` of length 20 where each element of `y` has a value 15. The execution takes a path P executing the loop 20 times, assigning the variable `x` to 110 and eventually covering δ at Line 11 . However, the program state after the execution of δ is not infected since after the first execution of δ, the value of `x` is 3 in both versions. We need not explore the subpaths after the execution of a changed region that does not cause any state infection if these subpaths do net lead to any other changed region.

Rationale P: Paths not propagating state infection to any observable output. Suppose δ is executed, the program state is infected after the execution of δ, but the infection does not propagate to an observable output. Let χ be the statement at which infection propagation stops. We need not explore the subpaths after the execution of χ if these subpaths do not lead to any other changed region.

2.2.2 Branching Nodes being Pruned

In DSE, path exploration is realized by flipping branching nodes. We next describe three categories of branching nodes that we avoid flipping corresponding to the preceding three categories of paths that we intend to avoid exploring.

Category E. This category contains all the branching nodes whose the other unexplored branch cannot lead to any changed region.

Category I. If a changed region is executed but the program state is not infected after the execution of the changed region, all the branching nodes after the changed region in the current execution path are included in this category.

Category P. Consider that a changed region is executed and the program state is infected after the execution of the changed region; However, the infection is not propagated to any observable output. Let χ be the last location in the execution path such that the program state is infected before the execution of χ but not infected after its execution. χ can be determined by comparing the value spectra [12] obtained by executing the test on both versions of the program. This category contains all the branching nodes after the execution of χ.

2.2.3 Branching Nodes being Prioritized

In addition to avoiding flipping the preceding categories of branching nodes, we prioritize branching nodes to be flipped after each run of DSE in the following two priority categories (*Priority 1* is the highest priority).

Priority 1. This category contains all the branching nodes that cannot lead to any changed region but the other unexplored branch of the branching node can lead to a changed region.

Priority 2. This category contains all the branching nodes whose both branches can lead to a changed region.

3 Preliminary Results

We prototyped part of our approach by manually inserting probes in program code to guide Pex [10] to avoid exploring branches in Categories E and I in the program code. We also inserted value-storing statements in the program code to compare the program states in the original and modified versions. In future work, we plan to completely implement our approach (including all the proposed pruning and prioritization) as a search strategy in Pex.

We conducted preliminary evaluation of our approach to assess its effectiveness. In our preliminary evaluation, we try to answer the following two research questions:

- **RQ1.** Can our approach more efficiently execute the changed regions between the two versions of a program than without using our approach?
- **RQ2.** Can our approach more efficiently infect the program states after the execution of changed regions than without using our approach?

To answer RQ1, we compare the number of runs of DSE required by the default search strategy in Pex with the number of runs required by our approach to execute a changed region. To answer RQ2, we compare the number of runs required by the default search strategy in Pex with the number of runs required by our approach to infect the program states after the execution of a changed region.

In our preliminary evaluation, we use the `tcas` program from the Software Infrastructure Repository (SIR) [3] as our subject. We converted the `tcas` program to C# (since the original `tcas` is written in C). We then seeded the first 11 faults available at SIR one by one to generate 11 new versions of `tcas`. We then used Pex to generate tests for `tcas`. To execute all these 11 changed statements in these 11 versions, the default strategy in Pex takes 115 runs in total, whereas our approach takes only 95 runs in total. The average and maximum percentages of run reduction for one version are 12.9% and 25%, respectively. To infect the program states after the execution of all these 11 changed statements, the default strategy in Pex takes 127 runs in total, whereas our approach takes 105 runs in total. The average and maximum percentages of run reduction for one version are 11.8% and 31.2%, respectively. The details of our preliminary results and the versions of `tcas` used in the preliminary evaluation are available at our project web page[2].

[2] `http://ase.csc.ncsu.edu/projects/regtestgen`

In summary, the preliminary evaluation answers the two research questions RQ1 and RQ2:

- **RQ1.** On average, our approach requires 12.9% fewer runs (maximum 25%) than the existing search strategy in Pex to execute the changed regions of the two versions of `tcas`.

- **RQ2.** On average, our approach requires 11.8% fewer runs (maximum 31.2%) than the existing search strategy in Pex to infect the program states after the execution of changed regions.

4 Related Work

Previous approaches [4, 9] generate regression unit tests achieving high structural coverage on both versions of the class under test. However, these approaches explore all the irrelevant paths, which cannot help in achieving any of the conditions P, I, or E in the PIE model. In contrast, we have developed a new search strategy for DSE to avoid exploring these irrelevant paths. In addition, our approach also prioritizes flipping of branching nodes to detect behavioral differences.

Santelices et al. [8] use data and control dependence information along with state information gathered through symbolic execution, and provide guidelines for testers to augment an existing regression test suite. Unlike our approach, their approach does not automatically generate tests but provides guidelines for testers to augment an existing test suite.

Some existing search strategies [2, 13] guide DSE to effectively achieve high structural coverage in a software system under test. However, these techniques do not specifically target to cover a changed region. In contrast, our approach guides DSE to avoid exploring paths that cannot help in executing a changed region. In addition, our approach avoids exploring paths that cannot help in P or I of the PIE model [11].

Differential symbolic execution [7] determines behavioral differences between two versions of a method (or a program) by comparing their symbolic summaries [5]. Summaries can be computed only for methods amenable to symbolic execution. However, summaries cannot be computed for methods whose behavior is defined in external libraries not amenable to symbolic execution. Our approach still works in practice when these external library methods are present as our approach does not require summaries. In addition, both approaches can be combined using demand-driven-computed summaries [1], which we plan to investigate in future work.

5 Discussion and Future Work

In our current approach, we perform DSE on the new version of a program. We then execute the test (generated after each run) on the original version. We can also perform DSE on the original version instead of the new version. In future work, we plan to conduct experiments to compare the effectiveness of the two approaches with respect to the types of changes.

Our current prioritization of branching nodes helps towards satisfying E of the PIE model. Currently, our approach does not prioritize branching nodes specifically toward satisfying I or P of the PIE model. In future work, we plan to prioritize branching nodes based on the probability to cause infection and to propagate the infection to an observable output. Moreover, we plan to develop additional priority categories based on data dependency from a changed region.

In future work, we plan to conduct experiments on systems with multiple changes and more complex systems to assess the benefit of our approach. For larger and more complex systems, the benefit of our approach is expected to be more substantial although the analysis cost of our approach would also increase in proportion to the size of systems.

Acknowledgments

This work is supported in part by NSF grant CCF-0725190 and ARO grant W911NF-08-1-0443.

References

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Proc. TACAS*, pages 367–381, 2008.

[2] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proc. ASE*, pages 443–446, 2008.

[3] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *ESE*, pages 405–435, 2005.

[4] R. B. Evans and A. Savoia. Differential testing: a new approach to change detection. In *Proc. FSE*, pages 549–552, 2007.

[5] P. Godefroid. Compositional dynamic test generation. In *Proc. POPL*, pages 47–54, 2007.

[6] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. *Proc. PLDI*, pages 213–223, 2005.

[7] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proc. FSE*, pages 226–237, 2008.

[8] R. A. Santelices, P. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Proc. ASE*, pages 218–227, 2008.

[9] K. Taneja and T. Xie. DiffGen: Automated regression unit-test generation. In *Proc. ASE*, pages 407–410, 2008.

[10] N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In *Proc. TAP*, pages 134–153, 2008.

[11] J. Voas. PIE: A dynamic failure-based technique. *TSE*, 18(8):717–727, 1992.

[12] T. Xie and D. Notkin. Checking inside the black box: Regression testing based on value spectra differences. In *Proc. ICSM*, pages 28–37, 2004.

[13] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. Technical Report MSR-TR-2008-123, Microsoft Research, Sep 2008.