Generating Test Data for Dynamically Discovering Likely Program Invariants

Neelam Gupta

Department of Computer Science
The University of Arizona
Tucson, AZ 85721
ngupta@cs.arizona.edu

Abstract

Dynamic detection of program invariants is emerging as an important research area with many challenging problems. As with any dynamic approach, the accuracy of dynamic discovery of likely program invariants depends on the quality of test cases used to detect invariants. Therefore, generating suitable test cases that support accurate detection of program invariants is crucial to the dynamic approach for invariant detection.

In this paper, we explore new directions in using the existing test data generation techniques to improve the accuracy of dynamically detected program invariants. First we discuss the augmentation of existing test suites to improve the accuracy of dynamically discovered invariants. The augmentation of the test suite may be done prior to running the dynamic analysis if the variables and expressions whose values will be monitored at runtime are known in advance. On the other hand, the dynamic analysis may be run first using an available test suite to obtain an initial guess of program invariants. These guessed invariants may then be used to generate test cases to augment the test suite. We also propose the use of existing test data generation techniques in improving the accuracy of invariants guessed using an already available test suite.

Keywords - *Test data generation, path testing, program invariants, dynamic analysis, execution traces.*

1 Introduction

Dynamic detection of program invariants is an emerging area of research with many challenging problems [3, 4]. The accuracy of dynamically discovered invariants critically depends upon the test suite used for detection of invariants. One parameter of the test suite that can be loosely related to the accuracy of dynamic detection of invariants is the size of the test suite. However, not all large test suites can be expected to be equally effective in accurate detection of invariants due to varying degree of structural coverage obtained. Thus, it is crucial to conduct research on what properties make a test suite suitable for dynamic invariant detection.

In prior work [3, 4], randomly generated and grammar generated test suites have been used for invariant detection. Randomly generated test suites have poor coverage and are most effective at highly peculiar bugs [10]. In the experiments reported in [4], the randomly generated test suites failed to execute many portions of a program. These randomly generated test suites did not detect many of the invariants that were detected using hand-crafted input cases. The experiments using randomly generated test suites from a grammar describing valid inputs detected more invariants than completely randomly generated test suites. However, generating test cases using grammar rules is a black box approach to test case generation and in general can fail to cover a significant part of the implementation.

In this paper we explore new research directions in generation of test cases to support dynamic invariant detection. We discuss the augmentation of existing test suites to improve the accuracy of dynamically discovered invariants. The augmentation of the test suite may be done prior to running the dynamic analysis if the variables and expressions, whose values will be monitored at runtime, are known in advance. On the other hand, the

dynamic analysis may be run first using an available test suite to guess program invariants. These guessed invariants may then be used to generate test cases to augment the test suite. We also propose the use of existing test data generation techniques to improve the accuracy of dynamically discovered likely invariants.

The organization of the paper is as follows. We discuss the background work in test data generation and dynamic detection of program invariants in section 2. In section 3, we propose new research directions to improve the accuracy of dynamically discovered invariants. Finally, we summarize the contributions of this paper and our future work.

2 Background

Test Data Generation Problem We consider the problem of generating input data that forces execution through a given path in a program. Symbolic evaluation [1, 2] and program execution based approaches [7, 8, 5, 11] have been proposed for generating test data for a given path in a program. The problem of test data generation for a given path is defined as follows.

> **Problem Statement:** *Given a program path P which is traversed for certain evaluations (true or false) of branch predicates $BP_1, BP_2 \cdots BP_n$ along P, generate a program input $I = (i_1, i_2 ..., i_m)$ in the input domain of the program that causes the branch predicates to evaluate such that P is traversed.*

The selection of paths for which the test input needs to be generated depends upon the testing strategy. For example, if the testing strategy is to ensure coverage of all branches in the program, the test paths are selected so that each branch is exercised by at least one test path among those selected.

Dynamic Invariant Detection. We consider the approach to dynamic discovery of invariants presented in [3, 4]. In this approach the invariants are dynamically detected from program traces that capture the variable values at program points of interest. The user runs the target program over a test suite to create execution traces of the program. An invariant detector determines which properties hold over both explicit variables and other expressions. Variable and expressions for which these properties hold over the traces, and also satisfy other tests such as being statistically justified, not being over unrelated variables and not being implied by other invariants, are reported as likely invariants. The set of likely invariants reported depends on the test suite used to discover invariants.

3 Test Data Generation for Dynamic Invariant Detection

In this paper we explore the relationship between the test data generation problem and dynamic discovery of program invariants. First we illustrate that the test suites satisfying the statement and branch coverage criteria may not be good enough for accurate detection of program invariants. We propose new approaches to to augment these test suites with additional test cases that can help in improving the accuracy of detected invariants. Second we illustrate the use of test data generation techniques in improving the accuracy of detected invariants.

3.1 Augmenting a Test Suite for Invariant Detection

We first illustrate the limitations of using existing structural coverage test suites for dynamic discovery of program invariants and propose how these test suites may be augmented with additional test cases to overcome these limitations.

```
0:   int funcEx(int x, y)
1:   {
P1:      if (x > 0)
2:           a=3;
3:           c=6;
4:       else
5:           a=3;
6:           c=9;
7:       endif
P2:      if (y > 0)
8:           b=4;
9:           d=2;
10:      else
11:          b=3;
12:          d=1;
13:      endif
14:  /* Monitored Property: (a*b == c*d) */
15:          printf(" a*b == c*d")
16:      ⋮
17:  }
```

Figure 1. An example code segment

Let us consider the code segment shown in Figure 1. Let the expression *(a*b == c*d)* in line 15 represent a property to be monitored during every ex-

ecution of this code segment. The code segment has been instrumented so that the value of this expression is written into every execution trace for this code segment. Let us say the test suite T_1 consists of the following two input cases.

$T_1 = \{(x = 5, y = 2), (x = -5, y = -1)\}$

Note that executing the code segment in Figure 1 with test cases in T_1 executes every statement in this code segment. In addition, every branch outcome of the two branch predicates $P1$ and $P2$ are executed by this test suite. Also note that every *definition-use* pair in this code segment is also exercised by this test suite. The property tested in line 15 will also hold for this test suite T_1. But it is easy to see that this property does not hold for the test case $(x = 5, y = -1)$. This simple example illustrates that code coverage (each statement executed at least once by some test case) and even branch coverage (each branch outcome is evaluated at least once by some test case) are very weak criteria for the test suite to be adequate for dynamic invariant detection.

However, the above example provides insight into the limitations of using coverage based test suites for detecting invariant properties at different points in the programs. These test suites are designed to test structural coverage of the program and may not contain test cases that are specifically helpful in verification of properties being monitored for invariant discovery. What is needed is the augmentation of these test suites with test cases specific to the properties being monitored.

In the above example, we need test cases for all possible combinations of branch outcomes by which the program execution can reach the critical point where the property of interest is being monitored. But in general, the number of paths reaching the critical point may be unbounded due to the presence of loops. So the crucial problem is *how to identify the important paths reaching the critical point* so that augmenting the structural coverage test suites with the test cases for these paths gives higher confidence in the value of the property being monitored during execution.

One approach we propose is to select the paths that exercise different *definition-use* pairs that are $live$ at the critical point where the invariant property is being monitored. However, in order to compute the *live definition-use* pairs at the critical point, we need to know the expression or the variable that is being monitored at this point. If the explicit variables and other expressions whose properties are collected in the executions traces are available in advance, then the *live definition-use* pairs for these variables and expressions can be computed.

On the other hand, if the explicit variables and expressions whose properties are to be monitored are not available in advance, then runtime analysis [3, 4] can be used to discover likely invariants with an existing structural coverage test suite. The *live definition-use* pairs for the discovered likely invariants (at the relevant program points) can then be used to guide the selection of paths important for verification of these discovered invariants. The test inputs for these paths can be generated and the structural coverage test suite can then be augmented with these test inputs. Now if the runtime analysis [3, 4] is done with the augmented test suite, it is expected that some of the spurious invariants that were reported earlier with the structural coverage test suite may not be reported any more. This is because the augmented test suite contains test cases specific to verification of those likely invariant properties that were reported earlier by the structural coverage test suite. The subset of the properties reported (from among those reported with the coverage test suite) by the augmented test suite is expected to be more accurate than the original set of likely invariants reported with the structural coverage test suite. We are currently exploring the effectiveness of this approach in our ongoing research. In the next section, we illustrate a different dimension of the relationship between the test data generation problem and the accuracy of reported program invariants.

3.2 Formulating Invariant Detection Problem as a Test Data Generation Problem

We propose to formulate the invariant detection problem as a data generation problem to improve the accuracy of dynamically discovered invariants. We illustrate this with the example in Figure 1. Let us replace line 15 in the code segment shown in Figure 1 by lines $P3$, 15, 16 and 17 shown in Figure 2.

We call the new branch predicate $P3$ introduced in the code segment in Figure 3 as the *invariant checking predicate*. Let us consider the problem of generating test data to execute the branch denoted by the line $P3$ followed by line 17, i.e., the *false* branch outcome of predicate $P3$. Now, *if test data can be generated for the false branch of an invariant checking predicate, then the corresponding property does not hold irrespective of the information collected from the execution traces using the already available test*

```
0:   int funcEx(int x, y)
1:   {
P1:      if (x > 0)
         .
         .
7:       endif
P2:      if (y > 0)
         .
         .
13:      endif
14:  /* Monitored Property: (a*b == c*d) */
P3:      if (a*b == c*d)
15:          printf("Property holds")
16:      else
17:          printf("Not an invariant")
         .
18:      .
19:  }
```

Figure 2. Modified example code segment

suites. As can be seen for the example in Figure 2, test data for the *false* outcome of $P3$ will be easily generated by program execution based techniques in [9, 12].

The above example illustrates an important application of the test data generation techniques in support of dynamic invariant detection. If test data can be generated to exercise the false branch of an invariant checking predicate, then the corresponding guessed invariant must be discarded. This is because this test input serves as a counterexample to this guessed invariant. Although, in general it is undecidable whether there exists an input to execute a given path in an arbitrary program, techniques [1, 2, 7, 8, 5, 11] have been developed for automatic generation of test data for a given path in a program. Different test data generation techniques have different strengths and the difficulty of test data generation for a path depends on the complexity and interdependence of branch predicates along the path. However, whenever test data generation techniques can generate an input exercising the *false* branch of an invariant checking predicate, the accuracy of the reported invariants can be significantly improved.

4 Conclusions and Future Work

In this paper we have provided insight into the relationship between test cases used for detecting invariants and the accuracy of invariant properties thus detected. We have proposed approaches for augmenting test suites for accurate detection of invariants. We are currently exploring these approaches for their effectiveness in accurate discovery of program invariants. We have also proposed the use of test data generation techniques to improve the accuracy of dynamically discovered program invariants.

References

[1] L.A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 3, pages 215-222, September 1976.

[2] R.A. DeMillo and A.J. Offutt, "Constraint-based Automatic Test Data Generation," *IEEE Transactions on Software Engineering*, Vol. 17, No. 9, pages 900-910, September 1991.

[3] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering,* vol. 27, no. 2, Feb. 2001, pp. 1-25.

[4] M. D. Ernst. "Dynamically Discovering Likely Program Invariants," *Ph.D. dissertation,* University of Washington Department of Computer Science and Engineering, (Seattle, Washington), Aug. 2000.

[5] M.J. Gallagher and V.L. Narsimhan, "ADTEST: A Test Data Generation Suite for Ada Software Systems," *IEEE Transactions on Software Engineering*, Vol. 23, No. 8, pages 473-484, August 1997.

[6] A. Gotlieb, B. Botella, and M. Rueher, "Automatic Test Data Generation using Constraint Solving Techniques," *International Symposium on Software Testing and Analysis*, 1998.

[7] N. Gupta, A. P. Mathur, and M. L. Soffa, "Automated Test Data Generation using An Iterative Relaxation Method" *ACM SIGSOFT Sixth International Symposium on Foundations of Software Engineering (FSE-6)*, pages 231-244, Orlando, Florida, November 1998.

[8] N. Gupta, A. P. Mathur, and M. L. Soffa, "UNA Based Iterative Test Data Generation and its Evaluation," *14th IEEE International Conference on Automated Software Engineering(ASE'99),* pages 224-232, Cocoa Beach, Florida, October 1999.

[9] N. Gupta, A. P. Mathur, M. L. Soffa, "Generating Test Data for Branch Coverage", *15th IEEE International Conference on Automated Software Engineering (ASE'00),* Grenoble, France, September 2000.

[10] D. Hamlet, "Random Testing," *Encyclopedia of Software Engg.,* 1994.

[11] B. Korel, "Automated Software Test Data Generation," *IEEE Transactions on Software Engineering*, Vol. 16, No. 8, pages 870-879, August 1990.

[12] B. Korel, A Dynamic Approach of Test Data Generation. In *Conference on Software Maintenance,* pages 311-317, San Diego, CA, November 1990.