

An Evaluation of the Effectiveness of Program Slicing in Reducing Duplication in Source Code

Rob McGregor and Will Thomasson

PROJECT SUMMARY

One major problem in software engineering is the lack of a method to create a test suite that will ensure perfect operation of the program under all possible inputs. Exhaustive testing can be impractical when there are an extremely high number of possible inputs, and it can in fact be infeasible in many systems where the combination of inputs is infinite. Incomplete testing of systems can and has led to many system failures, and in some cases has led to loss of human life.

This research seeks to find a method of determining the lack of code coverage of a test suite by using a program invariant detector, Daikon. Because the accuracy of the invariants output by Daikon depends on the completeness of the test cases, we apply Daikon to several programs for which the invariants are already known. If the invariants generated by Daikon differ from the known invariants, we can conclude that our test cases are insufficient. Also, we can use the output of Daikon to determine which test cases need to be added to the test suite in order to fully test the program. We can apply Daikon until we have developed a sufficient test suite.

Intellectual Merit

Writing test cases that accurately reflect the reliability of a system under all circumstances is difficult. If a given set of tests on a software system fails to test a part of the code that contains an error, then, once the system has been released, the error will likely occur sometime when the software is being used in practice. Therefore, we feel that it would be worthwhile to develop a method of determining how well a test suite is covering the code.

However, just knowing that a test suite is inadequate does not give much information. Simply adding random tests to an inadequate test suite is unlikely to dramatically improve the code coverage of the suite, if at all. Therefore, in addition to finding the coverage of the test suite, it is important to determine which tests or types of tests need to be added to the suite. Finally, once the new tests are added, it would be beneficial to see if the desired improvement in coverage was made.

This research addresses both of these issues by taking advantage of the program invariant detector Daikon. Daikon runs a program and its test cases and analyzes the running program to infer the program invariants. The accuracy of Daikon's results, however, relies on the test suite's ability to accurately portray all possible outcomes in the code. Inaccurate invariances will suggest an insufficient test suite, so by analyzing the inaccurate invariances, we will suggest a method to add test cases to the suite that will improve the accuracy of the invariants found. As the accuracy of the invariants continues to improve, an implied improvement in the test cases will result, providing a relatively easy method of improving software testing.

Broader Impact

By defining a methodology for improving test cases through invariant detection, we hope to provide a concrete process through which software engineers can develop better test cases for their programs. If successful, the methodology will enable software engineers to use their improved test cases to detect bugs in their programs that are often overlooked by the average test suite. This benefit would greatly improve reliability of programs.

1 Scientific Objectives

The objective of this research is to come up with an easy to use method of improving the quality of test suites for programs. We define the quality of the test suite as its ability to test the correct actions of the code under all possible circumstances. By improving the quality of a test suite, we produce a program that is less likely to have undetected errors.

Many programmers, and perhaps most, test many basic cases and maybe a few obscure cases that they can think of. They then run the program under these tests, and if it executes as they desired, they assume that the program works properly. The test suite could involve many tests under many different circumstances and different orders of code execution. Unless the test is exhaustive, though, meaning that every possible scenario has been tested, then there is no way to know for sure if the program contains a bug. It could be that the programmers overlooked an abnormal ordering of inputs that would not occur under normal circumstances, but could very well happen when used in the real world.

The problem with attempting to use exhaustive test cases is that it is often impractical, if not impossible, to guarantee that a test suite is exhaustive. Programmers are often willing to instead accept that there might be a few bugs rather than spend all of their time writing test cases. This is why it is beneficial to come up with a technique that can be used to improve test suites by locating potential shortcomings of existing suites, and thereby making clear what types of test cases need to be added.

Our objective is not to develop a perfect method of decompiling a suite that can guarantee perfect execution once all test cases have passed. This would not be a realistic goal. In fact, our research assumes that a somewhat reasonable test suite already exists for the program we are testing. Given the difficulty in analyzing the completeness of a test suite, and the fact that it is so difficult to determine appropriate tests to add to a suite in order to make it more complete, we only hope to remedy the situation. Our goal is to develop a useful method involving a program invariant detector that can determine the quality of a given test suite and give suggestions as to how to improve the quality by adding new, appropriate test cases.

2 Impact

By developing a well-defined methodology for improving test cases using invariant detection, we can provide a concrete process through which software engineers can develop superior test suites for their software. These test suites would exhibit efficiency, and would include an adequate number of test cases to cover all possible inputs, but not so many test cases that exhaustively testing them would be infeasible. With a superior test suite, the programmer can apply the test cases in the test suite to his program to find bugs that he or she may have missed using randomly selected test cases. By finding these rare bugs, the programmer can greatly increase the reliability of the program, an aspect that is paramount in certain applications, such as critical systems.

3 Research Plan

In this section, we will present the research goals and describe the tool. We will also describe the plan of research, and the method for developing improved testing suites.

3.1 Research Goals

The challenge of developing a sufficient test suite for a program can be tremendous. The test suite ideally must exhibit the behavior of all possible inputs. Because the number of possible inputs is exceedingly large, selecting a small subset to represent the rest is difficult, and exhaustively verifying that the subset does in

fact represent all the possible inputs is infeasible. A method of evaluating a test suite, and then of selecting additional test cases to improve the test suite if it is insufficient, would be a great help in determining whether a test suite is sufficient to cover nearly all possible cases.

3.2 Daikon Tool

Sufficiently detecting invariants cannot be done with static techniques alone. Pointers and other dynamically-allocated objects, for example, cannot be traced statically. Therefore, sufficient detection of invariants must involve analysis of the program while it is running. For this reason, Daikon executes the program and performs its analysis, examining associations between the values input as well as between the values computed in execution.

Because Daikon relies on the test cases provided to it, its detection of invariants is only as effective as the test cases. If test cases are limited in their scope of covering all possible inputs, the invariants will be limited in explaining the execution of the program. Similarly, if the set of test cases exhibits the behavior of all possible inputs, the invariants will have perfect accuracy in describing the execution of the program (of course, the latter is an ideal case that cannot realistically be achieved, but it is still a goal, one that can nearly be attained). This dependence on the test cases sets up a relation that can be used to examine the invariants given the test cases, or to examine the test cases given the invariants; we will perform the latter in our research.

The Daikon tool can be found online at <http://pag.lcs.mit.edu/daikon/>

3.3 Method

Our method of conducting research will involve several case studies. We will choose programs that have well defined invariants. The reason for this choice is that if we know all the invariants in advance, we will be able to more effectively use the Daikon invariant detector in order to analyze our test suite. An example of a program we will use is given in the paper “Dynamically Discovering Likely Program Invariants to Support Program Evolution” by Ernst, Cockrell, Griswold, and Notkin. According to the paper, the program is a “563 line C program with 21 procedures. It takes a regular expression and a replacement string as command-line arguments, then copies an input stream to an output stream while replacing any substring matched by the regular expression with the replacement string.”

Each program will begin with the code and two test suites. The first test suite should be adequate in that it will be making an attempt to test the accuracy of the program. This suite will be the focus of the majority of our research because our research is aimed at programmers who are looking to improve their existing test suites. Assumably, these programmers have at least made some effort to put together a reasonable test suite, but would like to see how well it covers the code. The second test suite will be similar to the first, but we will intentionally remove some tests from this suite which we know would allow a bug to enter the code untested. The reason for adding this second test suite is that it will be a sort of evaluation of our method to see how well it can detect an incomplete test suite when we know which parts of the test suite are incomplete. This is different from the first test suite, in which we may be able to determine tests that should be added, but had no prior knowledge that the test suite was incomplete.

Given code for a program and a test suite, we will run the program with the Daikon invariant detector. Daikon then comments the code with what it perceives to be the invariants of the program. We will then go through each of the invariants given by Daikon to make sure they are correct. We can ignore the invariants that are correct since our tests covered enough of the code for Daikon to come to the correct conclusion about a program invariant. It is the incorrect invariants that will allow us to determine which tests need to be added.

An example could be that Daikon gives us a program invariant saying that $x \geq 0$, whereas in actuality, there is no restriction on the value of x . In this case we would realize that we were testing only cases where x is positive, when in fact we should expand our test cases to ensure that negative values do not produce errors. This is a very simplified example, but the process will be the same for others. We will analyze each invariant; if it is incorrect or if we are unclear as to whether or not it is true, then we will have to analyze the program and the code to determine why an incorrect invariant would be assumed by Daikon in this particular case. For most of the incorrect invariants, we believe that it will be a lack of test cases that has caused the discrepancy, and we will add a test case, or possibly more than one if needed, that will attempt to show a contradiction to the invariant. These added test cases evaluate conditions that may not have been evaluated before their addition to the suite, therefore improving the quality of the test suite.

For those invariants about which we are uncertain, meaning it is questionable whether they are correct or not, one of two things can happen. First, after analyzing the program and the code, giving some thought to this invariant can lead us to the realization that it is actually true. Second, there are sure to be instances of invariants that we really cannot determine to be true or false. We feel that in this circumstance, it is acceptable to ignore the invariant, and move on, unless we can think of a test that could be added to contradict it.

Finally, there will be missing invariants. We may have a program invariant that should hold, but was not found by Daikon. If this is the case, we need to attempt to add tests that show this invariant is true.

Once we have analyzed all invariants and added test cases to our test suite to deal with them, we will repeat the cycle by running Daikon again with our new test suite. The goal is for each iteration to reduce the problems found with the invariants until we have determined that we can reduce them no more. At this point we will have our final set of test cases that we can achieve using our method.

4 Evaluation

The scientific objectives of this work are to improve the quality of test cases by using a program invariant detector. We realize that our method will be of little use in a system where the invariants are either very difficult to understand or are just not known for some reason. The reason for this shortcoming is that our method requires a general knowledge of the program invariants prior to running the invariant detector. However, we still feel that most programs will benefit from our analysis even if they do not have well defined invariants, or if the programmer has not taken the time to manually determine the invariants. The reason for this usefulness is that invariants related to critical aspects of the system are certain to be known unless the design is completely unclear. Daikon may output invariants that it assumes are correct; however, we may not know the correct invariants for the program. Nevertheless, there are always some basic invariants that will be understood by the design of the program, so we will still be able to benefit from Daikon's analysis by ensuring that these invariants are included in the output. We are aware that these will be some of the shortcomings of our method, so evaluating the success of our project will focus more on how well it works on systems where we will be sure to know the invariants.

We will run Daikon on programs that appear to have a good test suite and on programs that have a test suite that we know to be incomplete. The effectiveness of our method will be determined by how effective we are in analyzing the results of running Daikon with our code and test suites. If Daikon initially finds all program invariants for a given test suite, then we will assume that the test suite is adequate. However, if we know the test suite to be incomplete when we run Daikon, and it still finds all the invariants, then we will find this to be an obvious shortcoming of our method. We do not anticipate that this will be the case, however, given Daikon's reliance of its results on the test cases, and the fact that this was the basis of the idea of the method.

The cases where we anticipate our method to be evaluated best will be when Daikon gives invariants

that are incomplete or inaccurate. We will evaluate our ability to determine the test cases that will be added to the suite once we have received Daikon's output. Ideally, we should come up with at least one test case for each inaccurate invariant, unless it is the case that a given test case that we add will solve more than one of the invariant problems. Even if we do succeed in adding new test cases, they will only be successful additions if they result in a correction, or at least partial correction of invariants given by Daikon. We anticipate several iterations before the test suite will be determined complete, but it could be the case that we are unable to produce a test suite that has all the invariants that we expected. If this is the case, then it will present a weakness in our method as well. Our evaluation will take all of our programs that we tested into account and rate the usefulness of our method based on each of these cases.

5 Background

In this section, we describe the background ideas and theories used in this paper. This knowledge is useful for understanding the purpose behind our research.

5.1 Program Invariants

A key idea in program analysis is that of program invariants. Program invariants traditionally are detected by manually following the trace of a program, searching for patterns between the variables. For example, if $x=1$ and $y=1$ before a loop, and inside the loop are the statements $x=x+1$; and $y=2*y$; then the invariant over the loop is $y \geq x$. Another example is a linked list: for a node in the middle of the list, if the nodes after it in the list are never deleted, its pointer to the next node is never null.

Program invariants have many applications. One use includes using the invariants to trim unnecessary code; for example, if an invariant coincides with the test condition of an `if` statement, the `if` will always evaluate true, so the `if` can be eliminated). Another use is to implement the invariants as `assert` statements so that if changes are made to the code, the changes will not violate the invariants. Program invariants can also be inserted into code as comments to improve understanding of the code; this improved understanding is very beneficial for maintenance of legacy code. A further useful application of program invariants is evaluating test cases; for example, if an invariant states that $x < 100$, and all of the test values for x are less than 100, the test suite will not help in finding bugs with code involving the variable x . Upon realizing the inadequacy of the test suite, the programmer can add test cases, but more importantly, the programmer now knows exactly what sort of test cases to add. In the example above, the programmer needs to add a test case in which the value of x is greater than 100.

5.2 Test Suites

Another important idea in program analysis is the concept of test suites. Because programs often have very large numbers of inputs, exhaustively testing each input is infeasible. To test the program in a manner that covers all inputs, but is also efficient, requires the use of a test suite. The common notion of a test suite is a subset of all possible input, well-chosen to exhibit all the possible behaviors of all inputs. For example, if the `main` function in the program included the following code: `if(x mod 2 == 0) print x;` then the test suite need not contain all possible values of x ; it should contain two different values for x : one even, and one odd.

The test suite must be large enough that it represents all inputs during testing, but small enough that it can be exhaustively tested. Unfortunately choosing a test suite proves a challenge for all but the most trivial program.

6 Summary

Invariant detection can be applied to programs in order to improve their test suites. We have devised a method that uses a program invariant detection tool to examine programs with prespecified invariants, determining test cases that need to be added to the test suite in order to detect bugs that the test suite would otherwise have missed.