5L−6

Improving Fault Localization Based on Dynamic Slicing using Additional Assertions

Genki Sugimoto[1], Kazunori Sakamoto[2], Hironori Washizaki[1], and Yoshiaki Fukazawa[1]

[1]Waseda University
[2]National Institute of Informatics

Introduction

In software development, debugging is one of the most costly work for many developers. Due to the nature of debugging that it is likely to require longer time than being estimated, time spent on debugging is one major factor which prolongs software development. Furthermore, many developers say that debugging is tedious and like a duty. Thus, there is a strong need for making debugging easier.

There are various approaches to reduce the cost of debugging, and one of which is called dynamic slicing. The concept of dynamic slicing is to produce a slice, the set of statements that actually affect the value of a variable at a particular point for a particular execution, of a program. Applying this, for example, on malicious execution, one can focus on the lines which contribute to the malicious behavior, and thus debugging cost can be reduced.

There exist many tools and researches which succeed to reduce debugging cost using fault localization techniques. For example, as for effectiveness, [1] combined dynamic slicing with spectrum based fault localization technique. As for usability, [2] proposed visualization tool, named Tarantula, which visualizes results of spectrum based fault localization for easier interpretation. However, these have not yet solved the burden of debugging completely, and further improvements are required.

We proposes a new approach to improve the effectiveness of fault localization based on dynamic slicing using additional assertions. It allows developers to supply additional information on test executions so that the result of dynamic slicing can be refined more. Our approach adds interactive aspect in debugging on top of existing non interactive approaches.

Motivational Example

Our approach works well on top of the aforementioned Tarantula approach. Tarantula calculates suspiciousness, possibility of being faulty, of a statement s by following equation:

$$suspiciousness(s) = \frac{\frac{passed(s)}{totalpassed} + \frac{failed(s)}{totalfailed}}{\frac{passed(s)}{totalpassed}} \quad (1)$$

Where *passed(s)* stands for the number of passed test cases which executed statement s, *failed(s)* the number of failed test cases executed s, and *totalpassed* and *totalfailed* the total numbers of test cases that passed and failed, respectively.

While this result reflects the information obtained from the test execution results, it cannot reflect developers' knowledge about programs unless they add other test cases, which is not likely to occur in debugging process.

Our approach improves this in two ways. Firstly, while Tarantula calculates suspiciousness in the granularity of test cases, our approach focuses on every assertion statements in each test cases. Furthermore, it utilizes dynamic slicing of a variable in an assert statement rather than all statements executed by a test case like Tarantula. This can produce more accurate result. Secondly, it allows developers to add new assertions on arbitrary variables interactively. The concept is to manipulate the suspiciousness of statements by getting input from developers. For example, when a developer marked a variable v as correct, suspiciousness of statement s which did not contribute to the value of v is recalculated with incremented *totalpassed* value, and thus the suspiciousness is increased. Detailed architecture will be introduced in following chapter.

Overview of the Tool

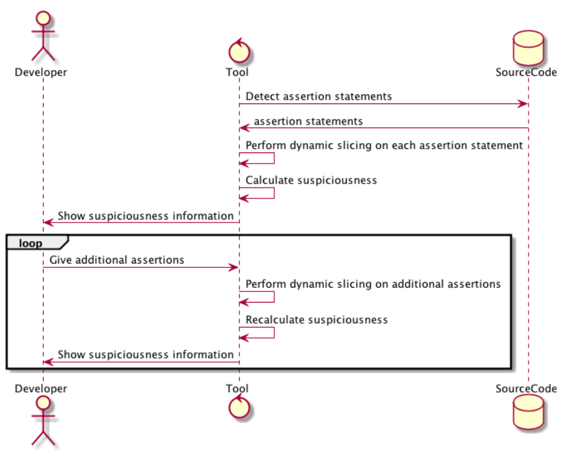
Developer
Tool
SourceCode
Detect assertion statements
assertion statements
Perform dynamic slicing on each assertion statement
Calculate suspiciousness
Show suspiciousness information
loop
Give additional assertions
Perform dynamic slicing on additional assertions
Recalculate suspiciousness
Show suspiciousness information
Developer
Tool
SourceCode

Figure 1: Tool Overview

We have created a prototype tool to implement our approach. The overview of the tool is shown in Figure 1. Before it starts interaction with a developer, it detects assertion statements in source code and its correctness. Then, it performs dynamic slicing on each assertion statements using a dynamic slicing tool named JavaSlicer [3], and calculates suspiciousness of each statement. After the suspiciousness is shown to the developer, it prompts developer to add additional assertion information. At this point, the developer can specify variables in the source code and its values. Then, new assertions for those variables and values are generated and evaluated, and the suspiciousness for each statement is refined. This interaction can be repeated as many times as the developer wants.

Our approach uses following equation to deal with cases like $totalpassed = 0$:

$$suspiciousness(s) = \frac{\frac{passed(s)+1}{totalpassed+1} + \frac{failed(s)+1}{totalfailed+1}}{\frac{passed(s)+1}{totalpassed+1}} \quad (2)$$

Case Study

```
 1  public class Statistic {
 2     public int sum;
 3     public double mean;
 4     public Statistic(List<Integer> l) {
 5        sum = 0;
 6        for (int value : l) {
 7           sum += value;
 8        }
 9        mean = sum / l.size();
10     }
11  }
12
13  public class StatisticTest {
14     @Test
15     public void testAll() {
16        List<Integer> v
17          = Arrays.asList(1, 2, 3, 4);
18        Statistic st = new Statistic(v);
19        assertTrue(st.mean == 2.5);
20     }
21  }
```

This example program contains a fault at line 9, which performs integer division while intending to obtain double type value. On the other hand, the calculation of `sum` at line 7 is not faulty.

Before any assertions are added by a developer, because *totalpassed, passed(7), passed(9), totalfailed, failed(7), failed(9)* are *0, 0, 0, 1, 1, 1* respectively, the suspiciousness calculated at lines 7 and 9 by formula (2) are both 2 as shown in Table 1 "Before" column.

The suspiciousness is 2 for both statements. Now, assume that a developer inputs a new assertion statement, `assertTrue(st.sum == 10)` before line 19. In that case, because the new assertion is correct and the dynamic slicing on the variable `st.sum` includes line 7 but not line 9, values of *totalpassed, passed(7), passed(9), totalfailed, failed(7), failed(9)* are now *1, 1, 0, 1, 1, 1* respectively, and the suspiciousness are recalculated and take values shown in Table 1 "After" column.

Table 1: Suspiciousness before and after adding an assertion

	Suspiciousness	
Line	Before	After
5	2	2
6	2	2
7	2	2
9	2	3

This shows that line 9, the faulty line, can be marked as more suspicious by our approach.

Conclusion and Future Work

We proposed a new approach which improves existing debugging approaches by 1) utilizing dynamic slices on each assertion statement rather than the sets of code executed by each test case and by 2) allowing developers to interactively and iteratively supply additional knowledges about the state of programs.

As a limitation, our approach is currently only effective on faults introduced in assignment statements, and not effective for faulty control statements. This problem will be managed in the future. Also, there are many possible improvements such as execution time and user interface. To reduce repeated input from developers, the feature of augmenting source code with interactively added assertions will be introduced as well.

References

[1] Birgit Hofer and Franz Wotawa. Spectrum Enhanced Dynamic Slicing for better Fault Localization. *ECAI*, pages 420–425, 2012.

[2] J.a. Jones, M.J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, (May):467–477, 2002.

[3] Andreas Zeller and Martin Burger. Design and Implementation of an Efficient Dynamic Slicer for Java submitted by. 2008.