Purifying Test Cases for Improving Fault Localization

Mr. M. Mohan¹ Akthar Nazir² Uzma Farooq³
¹Assistant Professor ²³M.Tech Student
¹,²,³Department of Computer Science & Engineering
¹,²,³SRM University

Abstract—Software fault localization is one of the most expensive, tedious and time consuming activities in program debugging. Therefore, there is a high demand for automatic fault localization that can guide programmers to the locations of faults, with minimal human intervention. This demand has led to proposal and development of various methods, each of which seeks to make the fault localization process more effective in its own unique and creative way. In this paper we use concept of test case purification for improving fault localization. The goal of test case purification is to separate existing test cases into small fractions called purified test cases, which consists of three major phases: test case atomization, test case slicing, and rank refinement, purified test cases consist of only one assertion statement in each test case.

Key words: Test case purification, test case atomization, test case slicing, rank refinement, assertion

I. INTRODUCTION

Locating bugs in a program is most tedious and time consuming activity which requires great effort on a part of programmer, to locate exact location of faults is called as fault localization. There is a need to make fault localization process effective in its own unique and creative way for which work is already going on. The goal of test case purification is to separate existing test cases into small fractions (called purified test cases) and to enhance the test oracles to further localize faults.

Test cases can be employed for fault localization [25], [17], [6]. Aborting the execution of a failing test case omits all the unexecuted assertions that are in the same test case. However, the effectiveness of fault localization depends on the quantity of test oracles.

We propose to use concept of test case purification for improving fault localization. The goal of test case purification is to generate purified versions of failing test cases, which include only one assertion per test case and excludes unrelated statements of this assertion. We leverage those purified test cases to better localize software faults in java projects. Test case purification for fault localization consists of three major phases: test case atomization, test case slicing, and rank refinement. First test case atomization generates a set of single assertion test cases for each failed test case; second, test case slicing removes the unrelated statements in all the failing single-assertion test cases; third, rank refinement combines the spectra of purified test cases with existing fault localization technique and sorts the statement as final result.

A test case also called test method is an executable piece of source code for verifying the behaviour of a software. In JUnit, a test case is formed as test method, which consists of two major parts, a test input and a test oracle. A test input is the input data to execute a program while a test oracle determines the correctness of the software with respect to its test input. An assertion is binary expression that indicates the expected behaviour of a program. If an assertion is not satisfied, an exception is thrown.

II. PROBLEM DESCRIPTION

A. Existing Technique

In the existing technique it had taken one particular test case into consideration and taken relevance of statements into consideration, and hence we need to debug one large test case which may consist of large number of lines of code and in case there is some fault in particular statement we need to check in and around of that statements which at times can be time consuming and involve higher cost for testing, which is an disadvantage.

B. Proposed Technique

We have taken above disadvantage into consideration and based on it we purified one particular test case into several purified versions of test cases by purified versions we mean smaller set of test cases, the advantage of splitting it into smaller fractions is that in case there is some fault or error in particular test case it will be easier to locate that fault as it will specify into which test case that fault is occurring and hence to correct it we will require less time as we need not to look into whole of a program but rather into that purified test case, and with respect to each test cases that returns ‘PASSED’ as status there is increase in percentage of test cases that are successfully executed with respect to number of test cases.

C. Design Diagram

The main aim of test case purification is to generate purified test cases from each failing test case. A purified test case is a smaller test case which consists of only one assertion and based on its status it can return pass or fail as status. We employ such purified test cases to improve existing techniques of fault localization.

Figure 1 depicts the overall design diagram of test case purification for fault localization. It consists of three major phases: test case atomization, test case slicing, and rank refinement.

In test case atomization each original failing test case with k-assertions is replaced by k single assertion test cases. A single assertion test case is a copy of the original test case, but only one of k original assertion is kept. In test case slicing each single assertion test case is treated as a program. We use dynamic method to remove irrelevant statements in each single assertion test case. Then short test cases are generated as purified test cases. In rank refinement, we give the percentage of test cases that passed or failed.
III. METHODOLOGY

A. Test Case Atomization

The goal of test case atomization is to generate a set of test cases for each failing test case. As the term atomization suggests, we consider each assertion has an atomic part in a test case. Given a failing test case with k assertions, we create k copies for this test case and we transform k – 1 assertions into regular test case statements for each copy (no exception from the assertion reaches the testing framework if the assertion fails). To transform an assertion into a regular test case statement, we surround this assertion with a try-catch structure shown in Figure 2.

In Java, the class java.lang.Throwable is a superclass of all the exceptions. As mentioned in, an exception will be thrown to the test case if an assertion is not satisfied. Based on the above structure, the exception will be caught as throwable and the test case will not be interrupted. Based on the surrounding structure in figure 2, a set of k single assertion test cases are created to replace each originally failing test case.

```java
try {
    /* assertion */
} catch (java.lang.Throwable throwable) {
    /* do nothing */
}
```

Fig. 2: A surrounding structure for transforming an assertion into a regular test case statement (no exception from the assertion reaches the testing framework if the assertion fails)

Note that in JUnit, two kinds of interruptions will stop the execution of a test case, namely a failure and an error. A failure is caused by an unsatisfied assertion, which is designed by developers; an error is caused by a fault, which is not considered by developers [10]. Thus, an error may appear in any statement of a test case. In test case atomization, we only deal with the failures (in assertions) in JUnit. If an error appears, the execution of a single-assertion test case will be aborted because an error usually causes severe problems, which are beyond the expected test cases by developers. After generating single-assertion test cases, we compile and execute all the single-assertion test cases. Meanwhile, we collect the failing ones among these test cases; for each failing single-assertion test case, we record its position that aborts the execution. This position is referred as a broken statement. For example, a broken statement in a single assertion test case could be an assertion (i.e., the exact assertion left in the test case) or a statement that throws an unexpected error. Finally, each failing single-assertion test case as well as its broken position is collected.

B. Test Case Slicing

The goal of test case slicing is to generate purified test cases before collecting their spectra. Given a failing single assertion test case resulting from test case atomization, we slice this test case by removing irrelevant statements. Program slicing can be mainly divided into two categories: static slicing [5] and dynamic slicing [18]. Informally, static slicing keeps all the possible statements based on static data and control dependencies while dynamic slicing keeps the actually executed statements in the dynamic execution (with dynamic data and control dependencies). In test case slicing, we use a dynamic slicing technique to remove statements in test cases since dynamic slicing may lead to more removal of statements [18]. In dynamic slicing, a slicing criterion should be specified before execution the program. A slicing criterion is defined as a pair < b, V >, where b is a statement in the object program and V is a set of variables to be observed at b. We perform dynamic slicing and slice single-assertion test cases during its execution by the JUnit framework. Our slicing criterion for a test case is its broken assertion with all the variables at this statement. Then we execute the dynamic slicing technique to collect the statements that will be removed. After the slicing, each failing single-assertion test case in test case atomization is updated with a purified test case. Then we execute these purified test cases on the project program and record the spectra for next phase.

C. Rank Refinement

The goal of rank refinement is to rank the test cases, that is if we have four test cases and one returns pass as status and three returns fail as status it will show pass status as 25%, similarly if we have two test cases that return pass as status and two as fail then it shows pass status as 50% which we have shown in our project.

D. Algorithm

**Input:** k-assertion test case T

**Output:** Set of purified single assertion test cases, T’

**Algorithm:**

1. **Step 1:** for p =1, 2, 3 ….,k, generate single assertion test cases, Tp
2. **Step 2:** for p =1, 2, 3 ….,k, test each single assertion in test case Tp to localize faults
3. **Step 3:** for p =1, 2, 3 ….,k, if assertion is TRUE, return ‘PASSED’ as status of test case Tp
4. **Step 4:** for p =1, 2, 3 ….,k, if assertion is not TRUE, return ‘FAILED’ as status of test case Tp
5. **Step 5:** Show the percentage of test cases with status ‘PASSED’ with respect to k number of test cases
6. **Step 6:** Generate test suite T’, with single assertion test cases t1, t2, t3, …tk
IV. RESULT ANALYSIS

Using the method of purification of test case, where in we generate smaller test cases which consists of single assertion statements, we broke down one particular test case into set of k smaller test cases, the advantage of splitting one particular test cases into smaller versions of test cases is that in case there is some fault or error in particular test case we need to look into that test case only hence it’s easy to localize fault in smaller set of test case rather than single large test case as it many consist of large number of lines of code.

Whenever test case has assertion that is true it returns test case “PASSED” as status, otherwise it will return “FAILED” as status, and based on number of test cases that returns “PASSED” as status there is increase in percentage of test cases that are successfully executed.

The graph shown in figure 3, depicts the result analysis of my project.

![Result Analysis Graph](image)

Fig. 3: Result Analysis graph showing increase in percentage of test cases successfully executed with respect to number of test cases.

V. RELATED WORK

We list the work related as follows:

A. Fault Localization Techniques

Fault localization aims to localize the faulty position in programs. Tarantula by Jones et al. [16] is an integrated framework to localize and visualize faults. Empirical evaluations of Tarantula on fault localization can be found in [15]. Abreu et al. [1] propose Ochiai and Jaccard for fault localization.

All of Tarantula, Ochiai, and Jaccard can be viewed as the state-of-art in spectrum-based fault localization. Naish et al. [19] propose a family of fault localization methods and empirically evaluate their results. Recent work by Zhang et al. [9] addresses the problem of how to identify faults with only failed runs. Xie et al. [19] propose a theoretical analysis on multiple ranking metrics of fault localization and divide these metrics into categories according to their effectiveness.

Hao et al. [13] propose a test-input reduction approach to reduce the cost of inspecting the test results. Gong et al. [11] design a diversity-maximization-speedup approach to reduce the manual labeling of test cases and improve the accuracy of fault localization. Yoo et al. [36] address the problem of fault localization prioritization. Their work investigates how to rank remaining test cases to maximize fault localization once a previous fault is found.

Baudry et al. [4] leverage the concept of dynamic basic blocks to maximize the ability of diagnosing faults with a test suite. Artzi et al. [2] directly generate test cases for localizing faults in invalid Html programs in dynamic web applications. This work does not require the test oracles since a web browser can report the crashes once invalid Html programs are found. Fault localization is also used as a phase of predicting a candidate position of the patch in software repair, such as GenProg [18] and Nopol [8].

Santelices et al. [21] combine multiple types of code coverage to find out the faulty positions in program. Baah et al. [3] employ potential outcome model to find out the dynamic program dependencies for fault localization. Xu et al. [17] develop a noise-reduction framework for localizing Java faults. This work is a general framework that can be used to improve multiple existing fault localization techniques. DiGiuseppe & Jones [9] recently propose a semantic fault diagnosis approach, which employs natural language processing to detect the fault locations. Xuan & Monperrus [22] develop a learning-based approach to combining multiple ranking metrics for fault localizing. Steinmann et al. [16] discuss the threats to validity in the empirical assessments of fault localization. Their work also presents the theoretical bounds of the accuracy in fault localization.

In our work, we address the same problem statement of fault localization. In contrast to existing work, test case purification is a framework to make better use of existing test cases. Our approach directly operates on test cases and can be generally applied to most of existing approaches.

B. Mutation and Slicing Based Fault Localization

Mutation-based fault localization has been recently proposed. The kernel idea of mutation-based fault localization is to localize faults by injecting faults. Zhang et al. [22] propose FIPL, a fault injecting approach to localizing faulty edits in evolving Java programs. Candidate edits are ranked based on the suspiciousness of mutants. Papadakis & Le Traon [27] develop Metallaxis-FIPL, a mutation-based technique for fault localization on C programs. Their work shows that test cases that are able to kill mutants can enable accurate fault localization. Moon et al. [23] recently propose MUSE, an approach based on both mutants of faulty statements and mutants of correct statements.

Slicing-based fault localization leverages program slicing to remove the statements in programs to find out the final faulty statements. Zhang et al. [18] employ dynamic slicing to reduce the size of C programs to avoid the distribution by irrelevant statements. Mao et al. [23] combine both statistic slicing and dynamic slicing to identify the faulty statements in programs. They empirically evaluate the slicing-based techniques on multiple fault localization techniques. Xie et al. [24] propose a new concept of metamorphic slice, based on the integration of metamorphic testing and program slicing. Metamorphic slices localize faults without the requirement of test oracles. Existing work on mutation-based and slicing-based fault localization aims to change the subject program to identify the faulty parts in the program. In our work, test case purification changes test cases for fault localization rather...
than subject programs. We make better use of existing test cases (test oracles) to improve the effectiveness of fault localization.

VI. CONCLUSION

In this paper, we propose a method for purifying test cases so as to improve fault localization, we split particular test case into several smaller fractions of test cases, called as purified test cases, these purified test cases consist of assertion statements, which if are TRUE return ‘PASSED’ as status otherwise return ‘FAILED’ as status, and there is increase in percentage of test case returning ‘PASSED’ as status with respect to total number of test cases, and the same have been shown in graph in Figure 3.

As future work, we will be further investigating the performance of our work on other java projects, we will also try to check for applicability of our work that is method of purifying test cases into other scenarios, where in we can save the time and cost to more extend.

ACKNOWLEDGEMENT

We conducted our work at SRM University, and we would take a note to thank our guide Mr. M. Mohan without whose contribution we would not have been able to complete our work, we would also like to thank Mr. A. Kulothungan whose support had immense impact on us in completing this work, we also thank everyone who was associated with us in one way or the other for their contributions.

REFERENCES


