A Brief Overview of Fault Localization

Student at East Carolina University, Greenville, NC, USA

Abstract — The world around us is frequently experiencing increasing software advances. With each of these advances, however, software defects are causing delays in the creation of such. Having faulty portions of a piece of software is an inevitable event. Moreover, due to the abstract and intangible nature of software, debugging - the process of removing defects - is a very difficult task, even for the most experienced software engineers. Therefore, since the beginning of software creation, software engineers have looked for ways to speed up the lengthy process of removing faults [9]. Both old and new techniques of debugging are discussed with an emphasis on the new sophisticated techniques. The individual types of automated debugging, also known as fault localization techniques, that are discussed in this paper are as follows: print tracing, assertions, breakpoints [9], program slicing [2], spectrum-based fault localization [3], similarity-based techniques - a branch of spectrum-based fault localization techniques [1], the Tarantula method - a similarity-based technique - and D* - another specific similarity-based technique [8].

I. INTRODUCTION

It is seen quite easily that nearly every single industry - financial, medical, aeronautical, and many more - rely heavily on some form of computer software to increase the advances in the field. With that being said, high quality software, that is software that produces adequate results, is crucial. In fact, a large portion of the time spent by a software developer is on debugging. One of the many problems with producing high quality software is the difficulty in tracking down faults quickly. The longer faults stay in a piece of software, the more of a negative impact the software has. For this reason, it is crucial for software engineers/computer scientists to develop methods for tracking down faults efficiently. Beginning not long after the creation of software, researchers started developing fault localization techniques. It was not until 2002 though, that the fault localization branch of software engineering experienced a huge spike in interest. Figure 1, from Wong et al., shows a clear rapid upwards trend of fault localization research papers published within the last 14 years [9].

Fault localization techniques have been around since software development began. The fault localization techniques used originally were okay for small programs and would even work for larger programs, but lacked speed and, more importantly, precision. A few of the original fault localization techniques were: Print tracing, Assertions, and Breakpoints [9]. Some of these old techniques are still in use but researchers are hopeful that they will be largely replaced by more accurate advanced techniques in the near future. Some of the categories of the newer techniques that have been developed are program slicing, spectrum-based fault localization, similarity-based techniques (a type of spectrum-based fault localization) [9]. Specific examples of similarity-based techniques that will be discussed in this paper are the Tarantula method [5] and the D* method [7]. All of the categories and specific techniques that were mentioned will be discussed in greater detail later in the paper.

II. TERMINOLOGY

For the convenience of the reader, a list of terminology will be given here to make the rest of the paper more comprehensible.

- fault, defect, and bug are all synonymous in this paper and refer to software not functioning as the developer intended.
- debugging will be the term used when referring to removing defects from a software system.
- Fault Localization Technique will be abbreviated as FLT.
- Spectrum-based fault localization will be abbreviated as SFL.
- Similarity-based technique will be abbreviated as SBT.
- Software Developers and Software Engineers are synonymous in this paper. The single term developer(s) will also refer to both of the aforementioned.

III. ORIGINAL FAULT LOCALIZATION TECHNIQUES

Original FLT's are techniques that have been used by software developers since the very early days of software creation. These techniques are the ones used by novice developers and developers who are working on small-scale software systems. For larger systems and more experienced developers, these techniques lack the ability to adequately identify the source of error(s). Although, they will hopefully be largely replaced by more sophisticated techniques in the future that do a better job in complex software systems, the original techniques will always be used because they are much easier to use and still just as effective when working with a small piece of software.

A. Print Tracing

Print Tracing (also known as Program Logging) is the act of using functions to display information to the screen. Generally, these functions can take string input that will be printed verbatim or they can take variable information which will be evaluated at run time. Typically, software engineers will simulate a portion of the code by hand, drawing diagrams
and other visual representations of computer memory, until they have an idea of where a bug might be located. Assuming their hand-simulation of the program narrows down the source of the bug, the developer will then add print statements all throughout the suspected error-prone region to identify where the execution is occurring (in the case of conditional execution) and to identify the value of variables/data structures present [9].

B. Assertions

Assertions are a form of conditional program execution where software developers use if statements to control the flow of execution. As a small example, if a developer thinks that a data structure should have a certain value just before a function call, and suspects that the data structure may not have proper value, an assertion could be used just before the function call. If the data structure has the proper value the function call would be executed, otherwise the function call would be skipped. In small software where the function call stack does not grow too large, such an assertion can be very helpful in determining if the function call is where the bug is or if the data structure value is where the bug is. [9].

C. Breakpoints

Breakpoints are a method of finding bugs that allows the developer(s) to pause the program at particular times during execution. One way breakpoints are used is to pause the program at a particular line of code that the developer may feel is causing problems. After pausing the program, the developer can start executing the program line by line to more likely allow them to see exactly where faulty behavior(s) occurs. Additionally, a breakpoint may be triggered on the change of a variable that the developer does not expect to change. For example, if a developer has a variable that they expect will remain unchanged in a block of code, they can set a breakpoint to trigger if that variable does in fact change in value. This gives a fairly good idea of what code is erroneous [9].

IV. SOPHISTICATED FAULT LOCALIZATION TECHNIQUES

Over the last 14 years a large number of research papers have been developed that discuss sophisticated FLTs. Sophisticated FLTs are those which use various statistical techniques on the information gathered through testing, irrelevant-code extraction, or other advanced ideas to hone in on defects [9]. These techniques are being developed continually and have shown to be more promising in helping developers when it comes to large-scale software systems. Program slicing, spectrum-based fault localization, and similarity-based techniques including the Tarantula method and D* will all be covered in detail in upcoming sections. Quite often, researchers use a well known test suite of programs

A. Program Slicing

Program slicing is a method whereby irrelevant pieces of code are removed from software making it easier for the software engineer to narrow down the location of faulty code. The method works by examining a variable value at a particular statement. All statements in the software that have no affect on the value of that variable at that statement are removed (sliced). After the slicing, only relevant statements to the variable of interest remain; this greatly reduces the amount of code the developer needs to examine. Program slicing is usually performed in one of two ways: statically or dynamically. Static slicing is slicing that is done by analysis of the program without the program being executed. Dynamic slicing, however, is slicing that is done by analysis of the program during the execution of the program [9].

One of the main concerns with the Program Slicing FLT is the size of a program slice. For the technique to be worthwhile,
the program slice needs to be a substantially smaller than original program so that the developer can quickly locate the bug. Static Slicing and Dynamic slicing both produce different average slice size results, although the difference is minor. Static slicing reduces the code to 26.9% of the original program size, in comparison with dynamic slicing which produced slices that were 26.3% of the original program size [2]. Both reductions reported were averages of numerous programs being sliced. The reason for a slightly larger average slice size in the case of static slicing is because the slice produced contains all possible statements that could affect the variable being observed at a particular statement. It should be clear that not all possible statements that could affect a variable are always executed, due to the nonlinear flow of execution with conditional execution control. The statements that could affect the variable and statement of interest that are not actually executed are usually able to be excluded in a dynamic slice because the dynamic slice takes into account the run-time behavior of the program [9]. Therefore, static slices are easier to produce because they do not need to consider the run-time behavior but the produce slightly larger slices, in comparison to dynamic slices which are slightly harder to compute but give a somewhat smaller slice.

B. Spectrum-Based Fault Localization

Spectrum-based Fault Localization (SFL) Techniques are those which use the information obtained during the testing of a program as the platform on which analysis is performed. The pieces of a program that are active during testing are known as the program spectra or the code coverage. There are program spectra for the test cases that a piece of software passes and for the test cases that a pieces of software fails. It used to be the case that program spectra were only used from the test cases that a piece of software failed on [9]. Logically, it seemed like a good idea to only use spectra from failed test cases since failed test cases are what reveal a bug to a developer in the first place. However, over time researchers began taking the spectra from passed test cases as well which led to more accurate fault localization. Furthermore, some researchers have extended SFLs further to categorize statements into two groups. Statements which are executed at least once during any of the failed test cases are put into the 'suspicious' group and statements which are never executed during any of the failed test cases are put into the 'unsuspicious' group. The regular SFL analysis is performed on the 'suspicious' group and the statements in the 'unsuspicious' group are immediately ranked the lowest in risk [10]. The Spectrum-based family of SFL techniques is quite large and includes, similarity-based techniques, statistics-based techniques, artificial intelligence based techniques, program-analysis based techniques, program entities, and more [3]. This paper will focus only on similarity-based techniques.

C. Similarity-based Techniques

Similarity-based techniques (SBTs) are a type of SFL technique that use metrics (statistical formulae) to determine the suspiciousness of program entities. Program entities are usually individual statements in a program but can also be other program components such as branches of execution, data structures, function calls, and more. When a SBT is used for fault localization, each program entity is assigned a score which ranks how likely that entity is to be faulty. The score assigned to the entity is determined based on a metric. The metric (also known as a similarity-coefficient sometimes) takes into account how many times a particular entity was covered during the passing cases and/or during the failing cases. The logic behind the metric is: entities which are executed more often in test cases that the program does not succeed on are more likely to be faulty entities than ones that are executed more often in test cases that the program does succeed on [1].

$$D^* \text{ Method: One SBT is the D^* Method. Created by Wong et al. The D^* method works by modifying a well known metric known as the Ochiai coefficient [8]. With its roots in molecular biology, the Ochiai coefficient has been used numerous times in fault localization papers as a way to rank statement suspiciousness. The D^* method is motivated from the fact that even though there are multiple methods to consider a statement suspicious or not suspicious, some of the methods should carry a higher way than others. A thorough listing of the intuitive methods for determining suspiciousness is as follows:}$$

- There should be a direct correlation between the suspiciousness of a statement and the number of failing test cases that execute the statement.
- There should be a inverse correlation between the suspiciousness of a statement and the number of passing test cases that execute the statement.
- The statements that are not executed during a failing test case are less suspicious than those that are executed during a failing test case.
- The direct correlation between the suspiciousness of a statement and the number of filing test cases that execute the statement should have more of an impact on the suspiciousness of a statement than the other two methods of determining suspiciousness.

The Ochiai coefficient takes into account all of the methods but does not weight the first method mentioned more heavily than the rest. For that reason, researchers modified the coefficient so that there would be more weight on the first method. This weight shifting led to a more accurate detection of bugs in software [8].

$$\text{Tarantula Method: A more well known SBT is the Tarantula Method. The Tarantula method is a similarity-based technique that makes use of the passed test cases in addition to the failed test cases. Allowing a statement which is primarily executed during failed test cases but occasionally executed during passed test cases allows for a more accurate prediction of where a bug is at. The reason for this is because in reality bugs follow this behavior, not the ideal case where a faulty statement is only executed in the failed test cases. The creators of the Tarantula method decided to use a coloring method for statements where the more red a statement is the higher suspiciousness it has and the more green a statement is the lower the suspiciousness it has. The coloring of a single statement s is determined by the equation:}$$

$$\text{hue}(s) = \frac{\text{passed}(s)}{\text{totalpassed}} + \frac{\text{failed}(s)}{\text{totalfailed}}$$

$$\text{(1)}$$
where passed(s) is the number of test cases that passed and executed s at least once. Likewise, failed(s) is the number of test cases that failed and executed s at least once. In the case that a denominator of a fraction is 0, the fraction gets assigned a value of 0 instead of undefined. The color produced by hue(s) for a statement directly determines how suspicious a statement is but in a nonintuitive way. In the case of hue(s), a value closer to 0 represents more suspiciousness and a value closer to 1 represents less suspiciousness. To account for this behavior the suspiciousness of a statement s is calculated as follows:

\[
suspiciousness(s) = 1 - hue(s)
\]  

Equation (2) leads to a more intuitive notion of suspiciousness where less suspicious values are closer to 0 and more suspicious values are closer to 1 [5].

For a better understanding of how tarantula works, Fig. 2 is provided. In Fig. 2 a function is presented which should calculate the middle number of three numbers passed to it. A series of test case values are provided to the function and execution of individual statements is recorded for passing and failing test cases.

Although Tarantula is one of the oldest techniques [3], it is still a very effective technique as shown by the study of James Jones and Mary Harrold in "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique" [5]. At the time Tarantula was studied by the previously mentioned researchers, there were few papers comparing the Tarantula method to other common methods at the time [4].

V. THE CHALLENGES OF FAULT LOCALIZATION TECHNIQUES

Although huge advances have been made in Fault Localization Techniques over the past decade, there is still doubt that automated techniques will sufficiently replace manual debugging. Many people in software engineering would say that manual debugging may be almost entirely impossible, if not entirely possible, to replace due to its creative aspects. Fault localization techniques, which make up a large portion of automated debugging techniques, operate under the assumption that a defect in code, can be precisely located [6]. However, in large scale, complex software systems, a defect can be nested within numerous statements of code that are very spread out. Another hesitance with FLTIs is the fact that they are very commonly tested on a test suite such as the Siemens suite. The majority of the test suites used contain injected faults, that is faults that are placed into the code on purpose. Inevitable injected faults do not always perfectly model the faults that occur accidentally while software is being developed.

For the reasons mentioned above, Lucia et al., performed analysis on three Java programs, all of which contained faults that formed accidentally during development. The faulty programs were found in a repository of faulty programs called iBugs and through JIRA (well known project planning and issue management tracking software). The first two programs that were collected through iBugs contained pre-bug code and post-bug code which could then be compared with advanced text difference tools. However, comments could have been added and code could have been moved around, both of which would not directly fix the bug but would be detected with a text difference analyzer. For that reason, the researchers decided to perform a manual analysis of the pre-bug code and the post-bug code to determine what was the cause of the bug in the first place. After analysis, across all 3 programs, it was found that 67% of faults were spread across more than one line of code. In addition to how many lines of code a bug was spread across the researchers also took note of how many methods and how many separate files a bug spanned. It was found that 56% of bugs spanned more than one method. Lastly, it was found that 27% of bugs spanned more than one file, although 88% of bugs were spread across no more than two files. Clearly, it is seen from Lucia et al. that faults are not as localizable and some FLTIs may assume [6].

VI. THE SUCCESS OF FAULT LOCALIZATION TECHNIQUES

In a similar discussion to the previous discussion, some researchers have studied how developers perform debugging tasks. Chris Parnin and Alessandro Orso performed a study on developers to see how they would accomplish a task with and without automated debugging tools [7].

VII. SUMMARY

This paper has provided a clear overview of the field of fault localization. Fault localization is crucial to software engineering because fault occurrences are inevitable and the process of tracing faults requires enormous amounts of time. Numerous types of fault localization were covered in this paper, each of which have advantages and disadvantages. Overall, the new and more sophisticated techniques are better performing for larger software systems and the older and less sophisticated techniques are better for small software systems. It should be clear that this field of research still has a lot that needs to be done including researching the effectiveness of each technique when used for one-fault systems versus many-fault systems. Additionally, more tests need to be developed for the FLTIs to be applied to. Modified versions of current methods are also being discovered quite frequently that prove to be more effective than the original versions. If the research in fault localization techniques continues, hi-quality software will be produced at a much higher rate than ever before.

REFERENCES

```c
mid() {
    int x, y, z, m;
    1: read("Enter 3 numbers:", x, y, z);
    2: m = z;
    3: if (y<z)
        4: if (x<y) ● ● ● ● 0.5 7
        5: m = y;
            ● 0.0 13
        6: else if (x<z) ● ● ● 0.71 2
        7: m = y; // *** bug *** ● ● 0.83 1
            else ● ● 0.0 13
    9: if (x>y) ● ● 0.0 13
    10: m = y;
        ● 0.0 13
    11: else if (x>z) ● 0.0 13
    12: m = x;
        0.0 13
    13: print("Middle number is:", m);
}
```

Fig. 2. A program showing the use of the Tarantula method to find a bug in a program that should calculate the middle number of three numbers.

---


