International Journal of Current Engineering and Technology
E-ISSN 2277 – 4106, P-ISSN 2347 – 5161

Available at http://inpressco.com/category/ijcet

*Research Article*

# Survey of Software Fault Localization for Web Application

**Swati B. Ghawate†* and Sharmila Shinde†**

†Department of Computer Engineering, JSCOE, Hadapsar, Pune, India

Accepted 20 April 2015, Available online 01 May 2015, **Vol.5, No.3 (June 2015)**

***Abstract***

*Fault localization or localizing the root cause of failure is one of the most difficult processes in software debugging. Hence, many automated techniques have emerged to help in this process. Most of these techniques are based on the principles used in real life for fault diagnosis. These techniques are based on statistical analysis of program constructs executed by passing and failing test case executions. Fault localization in dynamic web application is the problem of decisive where source code modification has to be completed in order to fix the perceived failures. The cause of the failure is called as execution bug that also called as fault. In the recent years automatic fault localization techniques are more demanding , that guide programmers to the locations of faults with minimal human intervention. Such high demand of fault localization led to development of various fault localization techniques. Although fault localization in general has been an active research topic, automatically localizing web faults has received very limited attention as of now. Therefore, in this paper we aim to understand existing fault localization techniques, we primarily focus on state of the art techniques and discuss some of the key issues and concerns that are relevant to fault localization.*

***Keywords:*** *Dynamic web application, Automatic Fault localization Techniques, Testing*

## 1. Introduction

Computer program may contain bugs regardless of the effort spent on developing it . As larger and more complex a program, the higher the chances of it containing bugs. Effectively and efficiently remove bugs in programs is always challenging for programmers, while not unknowingly introducing new ones at the same time. Furthermore, to debug, programmers must first be able to identify exactly where the bugs are, which is known as *fault localization*; and then find a way to fix them, which is known as *faultfixing*. In this paper, we focus only on fault localization.

Automated fault localization (AFL) techniques are developed to reduce the effort of software debugging, which is very frustrating task, often time-consuming and the costliest process in software development. Also finding root cause of a failure is the most difficult process in debugging. So techniques to automatically localize fault in software have come up.

Debugging can be divided into two main parts (H. Agrawal & R.A. DeMillo *et al*, 1993). The initial part is to recognize harmful code by using available testing technique. The next part is for programmers to actually examine the identified code to decide whether it certainly contains bugs. In the first part harmful code is prioritized based on its probability of containing bugs, the next part assumes that bug detection is perfect. All the fault localization techniques referenced in this paper focus on the first part, such that suspicious code is prioritized based on its likelihood of containing bugs. Code with a higher priority should be examined before code with a lower priority, as the former is more suspicious than the latter, i.e., more likely to contain bugs. As for the second part, we assume *perfect bug detection*, i.e., programmers can always correctly classify faulty code as faulty, and non-faulty code as non-faulty. If such perfect bug detection does not hold, then the amount of code that needs to be examined may increase.

Software fault localization for web application has not completely addressed yet. In this paper we referenced existing system Apollo (Shay Artzi *et al*, 2012) which localizes PHP based web application fault. WEB applications are typically written in a combination of several programming languages, such as Java-Script on the client side, and PHP with embedded Structured Query Language (SQL) commands on the server side. Such applications generate structured output in the form of dynamically generated HTML pages that may refer to additional scripts to be executed. When a failure is detected in web application, there is no HTML file or line number to point the developer.

*Corresponding author: **Swati B. Ghawate**

## 2. Problem Statement

Given a software that contains one or more faults, the objective of software fault localization is to localize code region that is most likely to contain fault. Here, some information about the bug may be initially present like a failing execution of the software, source code of the software, feedback from user about type of fault that occurs etc. Different techniques use different information about the fault. Given such information, the techniques pinpoint code regions that contain or are likely to contain the fault.

Failure represents a condition where the software either crashes or produces incorrect output in an execution of the software. A *fault / bug* represent code in the software that is the source of failure and thus needs to be modified. *Failing output* represents the location in the source code where failure is finally observed by the user. Hence, the aim of software fault localization is to locate code region that is likely to contain fault, given a failure or failing output. An execution trace represents the sequence of statements executed in the corresponding execution of the software. A failing trace corresponds to execution trace in an execution with failure and *correct / passing trace* represents an execution that is correct and does not show failure.

In traditional programming languages, the goal of fault localization is to find the faulty lines of code. Fault localization for web application is more difficult than this. Failures HTML code may be difficult to localize in the web application because HTML code is often dynamically generated by serverside code written, in PHP or Java and so, when a failure is detected, there is no HTML file or line number to point the developer to. Debugging and locating fault of web applications is very expensive and mostly manual task. When the developers observe an error in a web program either spotted manually or through automated testing techniques , the fault-localization process get started .

## 3. Fault Localization Techniques

One common way to locate bugs when a program execution fails is to insert *print* statements around the suspicious code. This approach adds the burden on programmers to decide where to insert *print* statements, as well as decide on which variable values to print. These choices are subjective, and may not be meaningful. And it is also not an ideal technique for identifying the locations of faults.

### *3.1 More Advanced Fault Localization Techniques*

Classification of fault localization techniques including, but not limited to, the following.

#### 3.1.1 Static, Dynamic, and Execution Slice-Based Techniques

Program slicing is a commonly used technique for debugging. Reduction of the debugging search domain via slicing is based on the idea that if a test case fails due to an incorrect variable value at a statement, then the bug should be found in the static slice associated with that variable-statement pair. Lyle & Weiser and H. Agrawal extended the above approach by constructing a program dice to further reduce the search domain for possible locations of a fault. A disadvantage of this technique is that it might generate a dice with certain statements which should not be included. To exclude such extra statements from a dice (as well as a slice), we need to use dynamic slicing instead.

An alternative is to use execution slicing and dicing to locate program bugs, where an execution slice with respect to a given test case contains the set of code executed by this test. There are two principles:

- The more successful tests that execute a piece of code, the less likely for it to contain any fault.
- The more that failed tests with respect to a given fault execute a piece of code, the more likely for it to contain this fault.

The problem of using a static slice is that it finds statements that could possibly have an impact on the variables of interest for *any* inputs instead of statements that indeed affect those variables for a *specific* input. Stated differently, a static slice does not make any use of the input values that reveal the fault. The disadvantage of using dynamic slices is that collecting them may consume excessive time and file space, even though algorithms have been proposed to address these issues. On the other hand, the execution slice for a given test can be constructed easily if we know the coverage of the test.

#### 3.1.2 Program Spectrum-based Techniques

A program spectrum records the execution information of a program in certain aspects such as how statements and conditional branches are executed with respect to each test. When the execution fails, such information can be used to identify suspicious code that is responsible for the failure. Tarantula (J. A. Jones and M. J. Harrold *et al*, 2005) is a popular fault localization technique based on the *executable statement hit* spectrum. It uses the execution trace information in terms of how each test covers the executable statements, and the corresponding execution result (success or failure) to compute the suspiciousness of each statement as X/(X+Y), where X = (number of failed tests that execute the statement)/(total number of failed tests), and Y = (number of successful tests that execute the statement)/(total number of successful tests). One problem with Tarantula is that it does not distinguish the contribution of one failed test case from another, or one successful test case from another. To overcome this problem, (Wong *et al*, 2010) propose that, with respect to a piece of code, the contribution of the *n*th failed test in computing its suspiciousness is larger than or equal to that of the (*n*+1)th failed test. The same applies to the contribution provided by successful tests. In addition, the total contribution of the failed tests is larger than that of the successful.

Renieris & Reiss propose a program spectrum-based technique, nearest neighbor, which contrasts a failed test with another successful test that is most similar to the failed one in terms of the distance between them. The execution of a test is represented as a sequence of basic blocks that are sorted by their execution counts. If a bug is in the difference set between the failed execution and its most similar successful execution, it is located. For a bug that is not contained in the difference set, the technique continues by first constructing a program dependence graph, and then including and checking adjacent un-checked nodes in the graph step by step until the bug is located. The set union, and set intersection techniques are also reported in (Renieris & Reiss *et al*, 2003)

3.1.3 Statistics-based Techniques

Several statistical fault localization techniques have also been proposed, such as Liblit05 (B. Liblit & M. Naik *et al* , 2005) and SOBER(C. Liu &L. Fei, X *et al*, 2006) which rely on the instrumentations and evaluations of predicates in programs to produce a ranking of suspicious predicates, which can be examined to find faults. However, these techniques are constrained by the sampling of predicates. They are also limited to bugs located in predicates, and offer no way to attribute a suspiciousness value to all executable statements. In light of such limitations, (Wong *et al*, 2008) propose a cross tabulation (crosstab) based statistical technique which uses only the coverage information of each executable statement, and the execution result with respect to each test case. It does not restrict itself to faults located only in predicates. More precisely, a crosstab is constructed for each statement with two column-wise categorical variables of covered, and not covered; and two row-wise categorical variables of successful execution, and failed execution. The exact suspiciousness of each statement depends on the *degree of association* between its coverage (number of tests that cover it) and the execution results.

3.1.4 Program State-based Techniques

A program state consists of variables, and their values at a particular point during the execution. A general approach for using program states in fault localization is to modify the values of some variables to determine which one is the cause of erroneous program execution. Zeller, *et al*. propose a program state-based debugging approach, delta debugging, to reduce the causes of failures to a small set of variables by contrasting program states between executions of a successful test and a failed test via their memory graphs. Based on delta debugging, Cleve & Zeller propose the cause transition technique to identify the locations and times where the cause of failure changes from one variable to another. A potential problem is that the cost is relatively high; there may exist thousands of states in a program execution, and delta debugging at each matching point requires additional test runs to narrow the causes. Another problem is that the identified locations may not be where the bugs reside. Gupta *et al*. 2005 try to overcome these issues by introducing the concept of failure inducing chops.

3.1.5 Machine Learning-based Techniques

Machine learning techniques are adaptive, and robust; and have the ability to produce models based on data, with limited human interaction. The problem at hand can be expressed as trying to learn or deduce the location of a fault based on input data such as statement coverage, etc. (Wong *et al*,2006) propose a fault localization technique based on a back-propagation (BP) neural network, which is one of the most popular neural network models in practice. The statement coverage of each test case, and the corresponding execution result, are used to train a BP neural network. Then, the coverage of a set of *virtual* test cases that each covers only one statement in the program are input to the trained BP network, and the outputs can be regarded as the likelihood of the statements being faulty. However, as BP neural networks are known to suffer from issues such as paralysis, and local minima, Wong *et al*. also propose an approach based on radial basis function (RBF) networks, which are less susceptible to these problems, and have a faster learning rate.

3.1.6 Fault Localization Tool Apollo For Dynamic Web application

Apollo (Shay Artzi *et al*, 2012) tool shows how the Tarantula, Ochiai, and Jaccard similarity coefficient faultlocalization algorithms can be enhanced to localize faults effectively in web applications written in PHP by using an extended domain for conditional and function-call statements and by using a source mapping. It also propose several novel test-generation strategies that are geared toward producing test suites that have maximal fault-localization effectiveness. Apollo implemented various fault localization techniques and test-generation strategies, and evaluated them on several open-source PHP applications.

It is also found that all the test-generation strategies that are considered are capable of generating test suites with maximal fault-localization effectiveness when given an infinite time budget for test generation. However, on average, a directed strategy based on path-constraint similarity achieves this maximal effectiveness after generating only 6.5 tests, compared to 46.8 tests for an undirected test-generation strategy

*3.1.7 Other Techniques*

There are other fault localization techniques including, but not limited to, data mining-based (P. Cellier& S.

Ducasse *et al*,2008) which discuss a combination of association rules and Formal Concept Analysis (FCA) to assist in fault localization), and model-based. Similarity-based coefficients such as Ochiai & Jaccard (R. Abreu, P. Zoeteweij *et al* , 2009 ; J. A. Jones & M. J. Harrold *et al* 2005 ) are used. Studies also examine the impact of coincidentally-successful tests on the effectiveness of fault localization techniques.

## 4. Important Aspects of Fault Localization Techniques

### *4.1 Effectiveness, efficiency, and robustness of a Fault Location Technique*

One important criterion to evaluate a fault localization technique is to measure its effectiveness in terms of the percentage of code, examined by programmers to locate the bug(s). In addition, a fault localization technique should also be efficient in that it should be able to present quality results in a reasonable amount of time without consuming extensive resources. Also, a test set when executed against the same program, but in two different environments, may result in two different sets of failed test cases. For a fault localizer relying on the coverage and test case execution results as its input, its effectiveness may therefore also vary depending on which environment it is employed in (or rather depending on which environment its input data is collected in). A fault localization technique should be robust to such variations in input (noise), and still perform effectively irrespective of environment.

### *4.2 Impact of Test Cases*

All empirical studies independent of context are sensitive to the input data. Similarly, the effectiveness of a fault localization technique also depends on the set of failed, and successful test cases employed. Using all the test cases to locate faults may not be the most efficient approach. Therefore, an alternative is to select only a subset of these tests. An important question that remains to be answered is how to select an appropriate set of test cases to maximize the effectiveness of a given fault localization technique.

### *4.3 Faults introduced by missing code*

One critique against all the fault localization techniques discussed is that they are incapable of locating a fault that is the result of missing code. However, the omission of the code may have triggered some adverse effect elsewhere in the program, such as the traversal of an incorrect branch in a decision statement. This abnormal program execution path may possibly assign certain code with unreasonably high suspicious values that provides a clue to programmers that some omitted code may be leading to control flow anomalies. Still, a more robust approach should be included in any fault localization technique to handle such faults.

### *4.4 Programs with multiple bugs*

The majority of current research on fault localization focuses on programs with a single bug. A possible extension to programs with multiple bugs can be achieved as follows. When two or more test cases result in a failed program execution, it is not necessary that all the failures are caused by the same fault(s). However, if there is a way to segregate or rather cluster failed executions together such that failed tests in each cluster are related to the same fault(s), then these failed tests, along with some successful tests, can be used to localize the corresponding causative fault(s). However, there are two significant challenges that need to be overcome. First, there may be more than one possible fault responsible for a failed execution. Second, a precise due to relationship between execution failures and causative fault(s) may not even be found without expensive manual investigation. Different clustering approaches have been proposed to address these challenges. However, significant research still needs to be done before such problems can be completely overcome.

## Conclusion

Choosing an effective debugging strategy usually requires expert knowledge regarding the program in question. In general, an experienced programmer's intuition about the location of the bug should be explored first.

We have seen different principles for fault localization and their application in various software techniques used for automatic fault localization. These principles are orthogonal to each other and cover different aspects of fault localization. Hence, we can explore combining these principles to build techniques that provide better fault diagnosis for web application.

However, even with the presence of so many different techniques, fault localization is far from perfect. While these techniques are constantly advancing, software too is becoming increasingly more complex, which means the challenges posed by fault localization are also growing. Thus, there is a significant amount of research still to be done, and a large number of breakthroughs yet to be made.

## References

R. Abreu, P. Zoeteweij, and A. J. C. van Gemund (2009), A Practical Evaluation of Spectrum-based Fault Localization, *Journal of Systems and Software*, 82(11):1780-1792

H. Agrawal, R. A. DeMillo, and E. H. Spafford (1993), Debugging with Dynamic Slicing and Backtracking, *Software – Practice & Experience*, 23(6):589-616

M. Renieris and S.P. Reiss (2003), Fault Localization with Nearest Neighbor Queries, Proc. IEEE Int'l Conf. Automated Software Eng.,pp. 30-39.

P. Cellier, S. Ducasse, S. Ferre, and O. Ridoux(2008), Formal Concept Analysis Enhances Fault Localization in Software,

*Proc. of the 4th International Conference on Formal Concept Analysis*, pp.273-288,

H. Cleve and A. Zeller (2005), Locating Causes of Program Failures, *Proc. of the 27th International Conference on Software Engineering*, pp. 342-351,

N. Gupta, H. He, X. Zhang, and R. Gupta (2005), Locating Faulty Code Using Failure-Inducing Chops, *Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering,* pp. 263-272.

J. A. Jones, J. Bowring, and M. J. Harrold (2007), Debugging in Parallel,*Proc. of the 2007 International Symposium on Software Testing and Analysis*, pp. 16-26.

J. A. Jones and M. J. Harrold (2005), Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique, *Proc. of the 20th IEEE/ACM Conference on Automated Software Engineering* . pp. 273-282

W. E. Wong, T. Wei, Y. Qi, and L. Zhao(2008), A Crosstab-based Statistical Method for Effective Fault Localization, *Proc. of the 1st International Conference on Software Testing, Verification and Validation*,pp. 42-51.

A. Zeller and R. Hildebrandt (2002), Simplifying and Isolating Failure-Inducing Input, *IEEE Transactions on Software Engineering*, 28(2):183-200.

Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia (2012) Fault Localization For Dynamic Web Applica-tion , IEEE Int'l Conf. Software Eng.,vol. 38

S. Artzi, J. Dolby, F. Tip, and M. Pistoia (2010), Practical Fault Localization for Dynamic Web Applications, Proc. 32nd ACM IEEE Int'l Conf . Software Eng., vol. 1, pp. 265-274.