A STUDY OF SOFTWARE FAULT LOCALIZATION TECHNIQUES

Anurag Negi¹, R.P. Mahapatra²

¹Mtech Student, Department of Computer Science and Engineering, SRM University (India)
²Head of Department, Department of Computer Science and Engineering, SRM University (India)

ABSTRACT
Fault localization is the activity of identifying the exact locations of program faults. Automatic software fault localization techniques are used by programmers to find out the exact location of the fault in least amount of time. Therefore, there is a high demand for automatic fault localization techniques that can guide programmers to the locations of faults, with minimal human intervention. This demand has led to the 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 article we provide an overview of several such methods and discuss some of the key issues and concerns that are relevant to fault localization.

Keywords: Software fault localization, program debugging, software testing, execution trace, suspicious code.

I. INTRODUCTION

No matter how much effort is spent on testing a program, it appears to be a fact of life that software defects are introduced and removed continually during software development processes. To improve the quality of a program, we have to remove as many defects in the program as possible without introducing new bugs at the same time.

During program debugging, fault localization is the activity of identifying the exact locations of program faults. It is a very expensive and time consuming process. Its effectiveness depends on developers’ understanding of the program being debugged, their ability of logical judgment, past experience in program debugging, and how suspicious code, in terms of its likelihood of containing faults, is identified and prioritized for an examination of possible fault locations.

There is a rich collection of literature that is abundant with various methods that aim to facilitate fault localization and make it more effective. While these methods share similar goals, they can be quite different from one another and often stem from ideas that themselves originate from several different disciplines.

There are basically two approaches for Software fault Localization:

A. Traditional Fault Localization Methods

To overcome the problem of fault finding, debugging tools such as DBX and Microsoft VC++ debugger have been developed. These tools allow users to introduce breakpoints along program execution and examine values of variables as well as internal states at each break point.
Users/programmers tend to use their experience, intuition and expertise along with the knowledge of the tool to find the bugs/faults in the program.

B. Advanced Fault Localization Methods

Fault localization can be divided into two major phases. The first part is to use a method to identify suspicious code that may contain program bugs. The second part is for programmers to actually examine the identified code to decide whether it actually contains bugs.

Further, the advanced fault localization techniques examine the code on the basis of degree of suspiciousness. It means that the code with higher degree of suspiciousness is examined first.

II. LITERATURE SURVEY

This section provides an overview of various Software Fault Localization methods that has been applied till now with their advantage and disadvantage:

A. Static, Dynamic and Execution Slice-Based Methods: Program slicing is a commonly used technique for debugging [1], [2]. A static program slice [3] for a given variable at a given statement contains all the executable statements that could possibly affect the value of this variable at the statement. 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 defect should be found in the static slice associated with that variable-statement pair. We can therefore confine our search to the slice rather than looking at the entire program.

One problem of any slicing-based method is that the bug may not be in the dice. And even if a bug is in the dice, there may still be too much code that needs to be examined. To overcome these problems, Wong et al. proposed an inter-block data dependency-based augmentation and refining method [4]. The former includes additional code in the search domain for inspection based on its inter-block data dependency with the code which is currently being examined, whereas the latter excludes less suspicious code from the search domain using the execution slices of additional successful tests. Different execution slice-based debugging tools have been developed and used in practice such as χSuds at Telcordia (formerly Bellcore) (5,6) and eXVantage at Avaya (7).

B. Program Spectrum-based Methods: A program spectrum records the execution information of a program in certain aspects, such as execution information for conditional branches or loop-free intra-procedural paths (8). It can be used to track program behavior (9). When the execution fails, such information can be used to identify suspicious code that is responsible for the failure. Early studies (10,11,12,13) only used failed test cases for fault localization, though this approach has subsequently been deemed ineffective (14,15,16). These later studies achieve better results using both the successful and failed test cases and emphasizing the contrast between them. The Executable Statement Hit Spectrum (ESHS) records which executable statements are executed. Two ESHS-based fault localization methods, set union and set intersection, are proposed in (17). The set union computes the set difference between the program spectra of a failed test and the union spectra of a set of successful tests. It focuses on the source code that is executed by the failed test but not by any of the successful tests. Such code is more suspicious than others. The set intersection method excludes the code that is executed by all the successful tests but not by the failed test. Renieris and Reiss (17) also propose another program spectrum-based method, nearest neighbor, which contrasts a failed test with another successful test which is most similar to the failed one in terms of the “distance” between them. In their method, the execution of a test is represented as
a sequence of statements that are sorted by their execution counts. If a bug is in the difference set, it is located. For a bug that is not contained in the difference set, the method can continue the bug localization by first constructing a program dependence graph and then including and checking adjacent un-checked nodes in the graph step by step until all the nodes in the graph are examined.

Another popular ESHS-based fault localization method is Tarantula (15) which uses the coverage and execution results 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. In (16), Wong et al. address two important issues: first, how can each additional failed test case aid in locating program bugs; and second, how can each additional successful test case help in locating program bugs. They propose that with respect to a piece of code, the contribution of the first failed test case that executes it in computing its likelihood of containing a bug is larger than or equal to that of the second failed test case that executes it, which in turn is larger than or equal to that of the third failed test case that executes it, and so on. This principle is also applied to the contribution provided by successful test cases that execute the piece of code.

A study on the Siemens suite (15) shows that Tarantula is more effective in locating a program bug, by examining less code before the first faulty statement containing the bug is identified, than another fault localization methods such as set union, set intersection, nearest neighbor (17) and cause transition techniques (18). Empirical studies have also shown that the method proposed in (16) is, in general, more effective than Tarantula.

Guo et al. (19) try to answer the question: during fault localization if a failed run (test case) is to be compared to a successful run, then which successful run should it be compared to? They do so by proposing a control flow-based difference metric that takes into account the sequence of statement executions in two runs instead of just the set of statement executions. Given a failed run and a pool of successful runs, they choose that successful run whose execution sequence is closest to the failed run based on the difference metric. Then, a bug report is generated by returning the difference between the sequences of the failed run and the successful run. Wong et al. (4) propose a more flexible approach by identifying successful tests that are as similar as possible to the failed test (in terms of their execution slices) in order to filter out as much code as possible. In this way, we start the fault localization with a very small set of suspicious code, and then increase the search domain, if necessary, using an inter-block data dependency-based augmentation method.

A few additional examples of program spectrum-based fault localization methods are listed below.

- **Predicate Count Spectrum (PRCS)-based**: PRCS records how predicates are executed. Such information can be used to track program behaviors that are likely to be erroneous. These methods are often referred to as statistical debugging because the PRCS information is analyzed using statistical methods. Fault localization methods in this category include Liblit05 (21), SOBER (22), etc. See “Statistics-based Methods” for more details.

- **Program Invariants Hit Spectrum (PIHS)-based**: This spectrum records the coverage of program invariants (23), which are the program properties that should be preserved in program executions. PIHS-based methods try to find violations of program properties in failed program executions to locate bugs. A study on the fault localization
using “potential invariants” is reported by Pytlik, et al. (24). The major obstacle in applying such methods is how to automatically find the necessary program properties required for the fault localization.

To address this problem, existing PIHS-based methods often take the invariant spectrum of successful executions as the program properties.

- Method Calls Sequence Hit Spectrum (MCSHS)-based: Information is collected regarding the sequences of method calls covered during program execution. For the purposes of fault localization, this data is helpful when applied to object-oriented software. In some cases, such a program may not fail even if the faulty code is executed; a particular sequence of method calls on the objects may also be required to trigger the fault. In one study, Dallmeier, et al. (25) collect execution data from Java programs and demonstrate fault localization through the identification and analysis of method call sequences. Both incoming method calls (how an object is used) and outgoing calls (how it is implemented) are considered. Liu et al. (26) construct software behavior graphs based on collected program execution data, including the calling and transition relationships between functions. They define a framework to mine closed frequent graphs from these behavior graphs and use them as a training set for classifiers that will identify suspicious functions.

C. Statistics-based Methods: Liblit et al. propose a statistical debugging algorithm (referred to as Liblit05) that can isolate bugs in the programs with instrumented predicates at particular points (21). Feedback reports are generated by these instrumented predicates. For each predicate $P$, the algorithm first computes $\text{Failure}(P)$, the probability that $P$ being true implies failure, and $\text{Context}(P)$, the probability that the execution of $P$ implies failure. Predicates that have $\text{Failure}(P) - \text{Context}(P) \leq 0$ are discarded.

Remaining predicates are prioritized based on their “importance” scores, which gives an indication of the relationship between predicates and program bugs. Predicates with a higher score should be examined first to help programmers find bugs. Once a bug is found and fixed, the feedback reports related to that particular bug are removed. This process continues to find other bugs until all the feedback reports are removed or all the predicates are examined.

Liu et al. propose the SOBER model to rank suspicious predicates (22). A predicate $P$ can be evaluated to be true more than once in a run. Compute $\pi(P)$ which is the probability that $P$ is evaluated to be true in each run as $\pi(P) = \frac{n(f)}{n(t) + n(f)}$ where $n(t)$ is the number of times $P$ is evaluated to be true in a specific run and $n(f)$ is the number of times $P$ is evaluated as false. If the distribution of $\pi(P)$ in failed runs is significantly different from that of $\pi(P)$ in successful runs, then $P$ is related to a fault. Wong et al. (29) present a crosstab (a.k.a. cross-classification table) analysis-based method (referred to as Crosstab) to compute the suspiciousness of each executable statement in terms of its likelihood of containing program bugs. A crosstab is constructed for each statement with twocolumn-wise categorical variables “covered” and “not covered,” and two row-wise categorical variables “successful execution” and “failed execution.” A hypothesis test is used to provide a reference of “dependency/independency” between the execution results and the coverage of each statement. However, the exact suspiciousness of each statement depends on the degree of association (instead of the result of the hypothesis testing) between its coverage (number of tests that cover it) and the execution results (successful/failed executions).
D. Program State-Based Methods: A program state consists of variables and their values at a particular point during the execution. It can be a good indicator for locating program bugs. A general approach for using program state-based 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 (30,31), 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 (32). Variables are tested for suspiciousness by replacing their values from a successful test with their corresponding values from the same point in a failed test and repeating the program execution. Unless the identical failure is observed, the variable is no longer considered suspicious.

Delta debugging is extended to the cause transition method by Cleve and Zeller (18) to identify the locations and times where the cause of failure changes from one variable to another. An algorithm named cts is proposed to quickly locate cause transitions in a program execution. A potential problem of the cause transition method 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 down the causes. Another problem is that the identified locations may not be the place where the bugs reside. Gupta et al. (34) introduce the concept of failure-inducing chop as an extension to the cause transition method to overcome this issue. First, delta debugging is used to identify input and output variables that are causes of failure. Dynamic slices are then computed for these variables, and the code at the intersection of the forward slicing of the input variables and the backward slicing of the output variables is considered suspicious.

Predicate switching (35) proposed by Zhang, et al. is another program state-based fault localization method where program states are changed to forcefully alter the executed branches in a failed execution. A predicate whose switch can make the program execute successfully is labeled as a critical predicate. The method starts by finding the first erroneous value in variables. Different searching strategies, such as Last Executed First Switched (LEFS) Ordering and Prioritization-based (PRIOR) Ordering, can be applied to determine the next candidates for critical predicates.

Wang and Roychoudhury (36) present a method that automatically analyzes the execution path of a failed test and alters the outcome of branches in that path to produce a successful execution. The branch statements whose outcomes have been changed are recorded as the bugs.

E. Machine Learning-based Methods: Machine learning is the study of computer algorithms that improve automatically through experience. Machine learning techniques are adaptive and robust and have the ability to producemodels based on data, with limited human interaction. This has led to their employment in many disciplines such as natural language processing, cryptography, bioinformatics, computer vision, etc. 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. It should therefore come as no surprise that the application of machine learning-based techniques to software fault localization has been proposed by several researchers.

Wong et al. (37) propose a fault localization method based on a back-propagation (BP) neural network which is one of the most popular neural network models in practice (38). A BP neural network has a simple structure, which makes it easy to implement using computer programs. At the same time, BP neural networks have the
ability to approximate complicated nonlinear functions (39). The coverage data of each test case (e.g., the statement coverage in terms of which statements are executed by which test case) and the corresponding execution result (success or failure) are collected. Together, they are used to train a BP neural network so that the network can learn the relationship between them. 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 (i.e., suspiciousness) of each statement containing the bug.

As BP neural networks are known to suffer from issues such as paralysis and local minima, Wonget al. (40) propose an approach based on RBF (radial basis function) networks, which are less susceptible to these problems and have a faster learning rate (41, 42). The RBF network is similarly trained against the coverage data and execution results collected for each test case, and the suspiciousness of each statement is again computed by inputting the coverage of the virtual test cases.

Briand et al. (43) use the C4.5 decision tree algorithm to construct a set of rules that might classify test cases into various partitions such that failed test cases in the same partition most likely fail due to the same fault. The underlying premise is that distinct failure conditions for test cases can be identified depending on the inputs and outputs of the test case (category partitioning). Each path in the decision tree represents a rule modeling distinct failure conditions, possibly originating from different faults, and leads to a distinct failure probability prediction. The statement coverage of both the failed and successful test cases in each partition is then used to rank the statements using a heuristic similar to Tarantula (15) to form a ranking based on each partition. These individual rankings are then consolidated to form a final statement ranking which can then be examined to locate the faults.

Brun and Ernst (28) build a learning model using machine learning (e.g., Support Vector Machines) to distinguish faulty and non-faulty programs using static analysis. General program properties (e.g., variables that are not initialized) are assumed to likely indicate the faults in programs and therefore in the learning model, properties of correct and incorrect programs are used to build the model. The classification step involves feeding as input the properties of a new program, and then the properties are ranked according to the strength of their association with faulty programs.

Ascari et al. (44) extend the BP-based method (37) by applying a similar methodology to Object-Oriented programs as well. They also explore the use of Support Vector Machines (SVMs) for fault localization.

G. Model-Based Methods: For model-based methods, the model used in each method is an important topic of research because the expressive capability of each model is crucial to the effectiveness of that method in locating program bugs.

DeMillo et al. propose a model for analyzing software failures and faults for debugging purposes (47). Failure modes and failure types are defined in the model to identify the existence of program failures and to analyze the nature of program failures, respectively. Failure modes are used to answer “How do we know the execution of a program fails?” and failure types are used to answer “What is the failure?” When an abnormal behavior is observed during program execution, the failure is classified by its corresponding failure mode. Referring to some pre-established relationships between failure modes and failure types, certain failure types can be identified as possible causes for the failure. Heuristics based on dynamic instrumentation (such as dynamic program slice) and testing information are then used to reduce the search domain for localizing the fault by predicting possible
faulty statements. One significant problem of using this model is that it is extremely difficult, if not impossible, to obtain an exhaustive list of failure modes because different programs can have very different abnormal behavior and symptoms when they fail. As a result, we do not have a complete relationship between all possible failure modes and failure types. This implies we might not be able to identify possible failure types responsible for the failure being analyzed.

Wotawa, et al. (47) propose to construct dependency models based on a source code analysis of the target programs to represent program structures and behaviors in the first order logic. Testcases with expected outputs are also transformed into observations in terms of first order logic. If the execution of the target program on a test case fails, conflicts between the test case and the models will be determined to find fault candidates. For each statement, a default assumption is made to suggest whether the statement is correct or incorrect. These assumptions will be revised during fault localization until the failure can be explained. The limitation is that their study only focuses on loop-free programs. To solve this problem, Mayer, et al. (48) present an approximatemodeling method in which abstract interpretation (49, 27) is applied to handle loops, recursive procedures, and heap data structures.

I. Data Mining-based Methods: Similar to machine learning, data mining also seeks to produce a model or derive a rule using relevant information extracted from data. Data mining can uncover hidden patterns in samples of data (which have been mined) that may not, and often will not, be discovered by manual analysis alone. Also sometimes the sheer volume of data that is available for analysis far exceeds that which can be analyzed by humans alone. Efficient data mining techniques transcend such problems and do so in reasonable amounts of time with high degrees of accuracy.

The software fault localization problem can be abstracted to a data mining problem. For example, we wish want to identify the pattern of statement execution that leads to a program failure. In addition, although the complete execution trace (including the actual order of execution of each statement) of a program collected during the testing phase is a valuable resource for finding the location of program faults, the huge volume of data makes it unwieldy to use in practice. Therefore, some studies have successfully applied data mining techniques, which traditionally deal with large amounts of data, to these collected execution traces.

Nessa et al. (33) generate statement subsequences of length $N$, referred to as $N$-grams, from the trace data. The failed execution traces are then examined to find the $N$-grams with a rate of occurrence higher than a certain threshold in the failed executions. A statistical analysis is conducted to determine the conditional probability that an execution fails given that a certain $N$-gram appears in its trace – this probability is known as the “confidence” for that $N$-gram. $N$-grams are sorted by descending order of confidence and the corresponding statements in the program are displayed based on their first appearance in the list. Case studies which apply this have shown that it achieves fault localization more effectively than Tarantula (15), by requiring the examination of less code before the first faulty statement is discovered.

Cellier et al. (37) discuss a combination of association rules and Formal Concept Analysis (FCA) to assist in fault localization. The proposed methodology tries to identify rules between statement execution and corresponding test case failure and then measures the frequency of each rule. Then, a threshold value is decided upon to indicate the minimum number of failed executions that should be covered by a rule to be selected. A
large number of rules so generated, can be partially ordered by the use of a rule lattice and then explored bottom up to detect the fault.

III. CONCLUSION AND FUTURE WORK

Locating program bugs is more of an art form than an easily-automated mechanical process. Although techniques do exist that can narrow the search domain, a particular method is not necessarily applicable for every program. Choosing an effective debugging strategy normally 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. However, if this fails, an appropriate fallback would be a systematic fault localization method based on solid reasoning and supported by case studies, rather than an unsubstantiated ad hoc approach.

Some fault localization methods are restricted to selecting only a single failed test case and a single successful test case, based on certain criteria, to locate a bug. Alternative methods rely on the combined data from sets of multiple failed and successful test cases. These latter methods take advantage of more test cases than the former, so it is likely that the latter are more effective in locating a program bug, in that they require the programmer to examine less code before the first faulty location is discovered. For example, the Tarantula method (15) which uses multiple failed and multiple successful tests, has been shown to be more effective than nearest neighbor (17), a method that only uses a single failed and single successful test. However, it is important to note that by considering only one successful and one failed test, it may be possible to align the two testcases and arrive at a more detailed root-cause explanation of the failure (18) compared to the methods that take into account multiple successful and failed test cases simultaneously. Neither category is necessarily superior to the other, but a general rule is that an effective fault localization method should assign higher suspiciousness to code that is likely to contain bugs and lower suspiciousness to code in which the presence of bugs is less probable. This increases the likelihood that the fault will appear near the top of the list when the code is prioritized for examination based on suspiciousness. An effective fault localization method should also, whenever possible, assign a unique suspiciousness value to each unit of code to reduce ambiguity during prioritization.

We would like to explore the machine learning based approaches because the learning rate of various machine learning approaches is dependent upon multiple parameters and therefore the optimization can be done to make these techniques more robust and effective.

We are currently focusing on modeling and optimizing GA-RBF Neural Network Algorithm (20) in order to use it as a fault localization technique. New algorithm takes longer running time in genetic algorithm optimizing, but it can reduce the time which is spent in constructing the network. Through the experiments analysis, the results show that the new algorithm greatly improves in generalization capability, operational efficiency, and classification precision of RBF neural network.

In conclusion, even with the presence of so many different fault localization methods, fault localization is far from perfect. While these methods 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


[20]. “A New Optimized GA-RBF Neural Network Algorithm” Weikuan Jia, Dean Zhao, Tian Shen, Chunyang Su, Chanli Hu, and Yuyan Zhao, Computational Intelligence and Neuroscience Volume 2014 (2014), Article ID 982045, Hindawi 13 October 2014


