A Survey of Software Fault Localization

Technical Report UTDCS-45-09

Department of Computer Science
The University of Texas at Dallas

November 2009

W. Eric Wong and Vidroha Debroy
{ewong,vxd024000}@utdallas.edu

ABSTRACT

Software fault localization is one of the (if not the) most expensive, tedious and time consuming activities in program debugging. 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.

KEY WORDS: Software fault localization, program debugging, software testing, execution trace, suspicious code

1. INTRODUCTION

No matter how much effort is spent on testing a program,[1] 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. No article, regardless of breadth or depth can hope to cover all of them. Therefore, while we aim to cover as much ground as possible, we primarily focus on the state of the art techniques in the area of fault localization and try to discuss them in as much detail as possible as we go along. The remainder of this article is organized in the following manner: we begin by describing some fundamental and intuitive fault localization methods in Section 2, and subsequently move on to more advanced and complex ones in Section 3. Then in Section 4 we discuss some of the key issues and aspects pertinent to the area of fault localization, and finally present the conclusions in Section 5.

[1] We use "bugs," "faults" and "software defects" interchangeably. We also use "program," "application" and "software" interchangeably. In addition, "locating a bug" and "localizing a fault" have the same meaning, and "a statement is covered by a test case" and "a statement is executed by a test case" are used interchangeably.

2. TRADITIONAL FAULT LOCALIZATION METHODS

One intuitive way to locate bugs when a program execution shows some abnormal behavior is to analyze the corresponding memory dump. Another way is to insert *print* statements around suspicious code to print out the values of some variables. While the former is not often used now because it might require an analysis of a tremendous amount of data, the latter is still used. However, users need to have a good understanding of how a program is executed with respect to a given test case that causes the trouble and then insert only the necessary (i.e., neither too many nor too few) print statements at the appropriate locations. As a result, it is also not an ideal debugging technique for identifying the locations of faults.

To overcome the problems discussed above, debugging tools such as DBX and Microsoft VC++ debugger have been developed. They allow users to set break points along a program execution and examine values of variables as well as internal states at each break point, if so desired. One can think of this approach as inserting print statements into the program except that it does not require the physical insertion of any print statements. Break points can be pre-set before the execution or dynamically set on the fly by the user in an interactive way during program execution. Two types of execution are available: a continuous execution from one break point to the next break point, or a stepwise execution starting from a break point. While using these tools, users must first decide where the break points should be. When the execution reaches a break point, they are allowed to examine the current program state and to determine whether the execution is still correct. If not, the execution should be stopped, and some analysis as well as backtracking may be necessary in order to locate the fault(s). Otherwise, the execution goes on either in a continuous mode or in a stepwise mode. In summary, these tools provide a snapshot of the program state at various break points along an execution path. One major disadvantage of this approach is that it requires users to develop their own strategies to avoid examining too much information for nothing. Another significant disadvantage is that it cannot reduce the search domain by prioritizing code based on the likelihood of containing faults on a given execution path.

3. 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 indeed contains bugs. All the fault localization methods referenced in the following 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. A bug in a piece of code (e.g., a statement) will be detected by a programmer if the code (namely, the statement) is examined. If such perfect bug detection does not hold, then the code (the number of statements in this case) that needs to be examined in order to find the bug may increase. Without loss of generality, hereafter code may be referred to as statements with the understanding that fault localization methods can also be applied to identify suspicious functions, decisions, def-uses, etc.

3.1. Static, Dynamic and Execution Slice-Based methods

Program slicing is a commonly used technique for debugging (60,65). A static program slice (64) 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. Lyle and Weiser extended the above approach by constructing a program dice (as the set difference of two groups of static slices) to further reduce the search domain for possible locations of a fault (43). A disadvantage of this method is that it might generate a dice with certain statements which should not be included. This is because we cannot predict some run-time values via a static analysis. To exclude such extra statements from a dice (as well as a slice), we need to use dynamic slicing (3,34) instead of static slicing as the former can identify the statements that indeed do, rather than just possibly could as by the latter, affect a particular value observed at a particular location. Studies such as (1,18,33,42,58,81,82) which use the dynamic slicing concept in program debugging have been reported.

An alternative is to use execution slicing and dicing based on dataflow tests to locate program bugs (4) where an execution slice with respect to a given test case contains the set of statements executed by this test. The reason for choosing execution slicing over static or dynamic slicing is that a static slice focuses on finding 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. This implies that a static slice does not make any use of the input values that reveal the fault. It clearly violates a very important concept in debugging which suggests programmers analyze the program behavior under the test case that fails and not under a generic test case. The disadvantage of using dynamic slices is that collecting them may consume excessive time and file space even though different algorithms (9,24,36,79,80) have been proposed to address these issues. On the other hand, the execution slice for a given test case can be constructed very easily if we know the coverage of the test because the corresponding execution slice of the test can be obtained simply by converting the coverage data collected during the testing into another format, i.e., instead of reporting the coverage percentage, it reports which statements are covered.

A study examining the execution dice of one failed and one successful test for locating program bugs was reported in (4). Wong et al. (71) extended that study by using multiple successful and failed tests based on the following observations:

- The more that successful tests 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.

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 (68). 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) (5,84) and eXVantage at Avaya (67).

3.2. 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 (26). It can be used to track program behavior (56). When the execution fails, such information can be used to identify suspicious code that is responsible for the failure. Early studies (2,33,35,59) only use

failed test cases for fault localization, though this approach has subsequently been deemed ineffective (4,31,66). 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 (55). 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 (55) 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 (31) 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 (66), 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 (31) 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 other fault localization methods such as set union, set intersection, nearest neighbor (55) and cause transition techniques (15). Empirical studies have also shown that the method proposed in (66) is, in general, more effective than Tarantula.

Guo et al. (25) 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. (68) 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 (38), SOBER (39), etc. See "Statistics-based Methods" for more details.

- Program Invariants Hit Spectrum (PIHS)-based: This spectrum records the coverage of program invariants (20), 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. (54). 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. (17) 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. (41) 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.

Other program spectra such as those in Table 1 (26) can be used to help identify suspicious code in the program.

Table 1. Sample program spectra for fault localization

	Name	Description
BHS	Branch Hit Spectrum	conditional branches that are executed
CPS	Complete Path Spectrum	complete path that is executed
PHS	Path Hit Spectrum	intra-procedural, loop-free path that is executed
PCS	Path Count Spectrum	number of times each intra-procedural, loop-free path is executed
DHS	Data-dependence Hit Spectrum	definition-use pairs that are executed
DCS	Data-dependence Count Spectrum	number of times each definition-use pair is executed
OPS	Output Spectrum	output that is produced
ETS	Execution Trace Spectrum	execution trace that is produced

3.3. 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 (38). Feedback reports are

generated by these instrumented predicates. For each predicate P, the algorithm first computes *Failure*(P), the probability that P being true implies failure, and *Context*(P), the probability that the execution of P implies failure. Predicates that have *Failure*(P) – *Context*(P) ≤ 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 (39). 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 $\pi(P) = \frac{n(t)}{n(t)+n(f)}$ 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. (72) 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 two column-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).

The major difference between Crosstab, SOBER, and Liblit05 is that Crosstab ranks suspicious statements, whereas the last two rank suspicious predicates for fault localization. For SOBER and Liblit05, the corresponding statements of the top k predicates are taken as the initial set to be examined for locating the bug. As suggested by Jones and Harrold in (31), Liblit05 provides no way to quantify the ranking for all statements. An ordering of the predicates is defined, but the approach does not expand on how to order statements related to any bug that lies outside a predicate. For SOBER, if the bug is not in the initial set of statements, additional statements have to be included by performing a breadth-first search on the corresponding program dependence graph, which can be time consuming. However, this search is not required for Crosstab, as all the statements of the program are ranked based on their suspiciousness. An extension of a recent study (72) reports that Crosstab is almost always more effective in locating bugs in the Siemens suite than SOBER and Liblit05.

3.4. 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 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 (75,76), 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 (77). 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 (15) 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. (23) 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 (78) 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 (61) 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.

3.5. 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 produce models 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. (69) 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 (21). 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 (27). 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, Wong et al. (70) propose an approach based on RBF (radial basis function) networks, which are less

susceptible to these problems and have a faster learning rate (37,63). 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. (11) 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 (31) 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 (12) 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. (6) extend the BP-based method (69) by applying a similar methodology to Object-Oriented programs as well. They also explore the use of Support Vector Machines (SVMs) for fault localization.

3.6. 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 (19). 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. (74) 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. Test cases 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 approximate modeling method in which abstract interpretation (10,16) is applied to handle loops, recursive procedures, and heap data structures. In addition, abstract interpretation is also used to improve the accuracy of model-based diagnosis methods (49). Wotawa (73) also demonstrates that both static and dynamic slices can be equivalent to conflicts in the model-based diagnosis methods for fault localization.

Mateis, et al. (45) present their functional dependency model for Java programs that can handle a subset of features for the Java language, such as classes, methods, assignments, conditionals, and while-loops. In their model, the structure of a program is described with dependency-based models while logic-based languages, such as first order logic, are applied to model the behaviors of the target program. The dependency-based model is then extended to handle unstructured control flows in Java programs (46,47), such as exceptions, recursive method calls, *return* and *jump* statements. In additional to functional dependency models, value-based models (32,50) that represent data-flow information in Java programs are also applied to localize components that are responsible for the difference between program behaviors and expected behaviors. An experimental project named JADE (the Java Diagnosis Experiments) (44,51) is used to demonstrate the applicability of model-based diagnosis in software debugging. The project integrates various models and the model-based diagnosis engine into an interactive debugging environment with a graphical user interface.

3.7. 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. (52) 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 method to the Siemens suite (29) as well as space and grep (57) have shown that it achieves fault localization more effectively than Tarantula (31), by requiring the examination of less code before the first faulty statement is discovered.

Cellier et al. (14) 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.

4. IMPORTANT ISSUES AND RESEARCH AREAS

In this section we explore some critical aspects related to software fault localization that are important research areas in and of themselves. Should this article manage to inspire the reader enough to develop a fault localization strategy of their own, then these are important concerns that need to be taken into account and addressed. Also as mentioned before in Section 3, while this discussion is focused on statements, it is implicit that fault localization can also be accomplished with respect to decisions, def-uses, etc.

4.1. Effectiveness of a Fault Location Method

The effectiveness of a fault localization method can be measured by a score $\mathcal{EXAM}$ in terms of the percentage of statements that *have to be examined* until the first statement containing the bug is reached (66,69,70,72). A similar score using the percentage of the program that *need not* be examined to find a faulty statement is used in (15,31,55). These two scores provide the *same* information, but the $\mathcal{EXAM}$ score is more direct and easier to understand. The effectiveness of different fault localization methods can be compared based on $\mathcal{EXAM}$. Although a bug may span multiple statements which may not be contiguous or even multiple functions, the fault localization stops when the first statement containing the bug is reached. This is because the focus is to help programmers find a *starting* point to fix a bug rather than provide the complete set of code that has to be modified/deleted/added with respect to each bug.

As discussed in the beginning of Section 3, we assume perfect bug detection, that is, a bug in a statement will be detected by a programmer if the statement is examined. If it is not the case, then the number of statements that need to be examined in order to find the bug may increase and the effectiveness of the fault localization method will decrease.

Different statements may be assigned the same suspiciousness by a fault localization method. If this happens, it gives two different types of effectiveness: the "best" and the "worst." The "best" effectiveness assumes that the faulty statement is the first to be examined among all the statements of the same suspiciousness. For instance, supposing there are ten statements of the same suspiciousness of which one is faulty, the "best" effectiveness is achieved if the faulty statement is the first to be examined of these ten statements. Similarly, the "worst" effectiveness occurs if the faulty statement is the last to be examined of these ten statements. An effective fault localization method should ideally assign a unique suspiciousness value to each statement.

For a faulty program P, we consider the $\mathcal{EXAM}$ score of methods $\mathcal{M}_1$ and $\mathcal{M}_2$. If the score of $\mathcal{M}_1$ for locating all the bugs in P is smaller than that of $\mathcal{M}_2$, then fewer statements must be examined

by the programmer to locate these bugs using method $\mathcal{M}_1$ compared to $\mathcal{M}_2$. Thus, $\mathcal{M}_1$ is more effective than $\mathcal{M}_2$ for locating bugs in P.

4.2. Clustering for Programs with Multiple Bugs

Many fault localization methods discussed in Section 3 are focused on a *single* bug, i.e., each faulty program has exactly one bug. Hence, if a program execution fails, the cause of the failure is clear.

For a program with multiple bugs, one approach is to follow the one-bug-at-a-time strategy. After a bug is found and fixed, the modified program has to be tested again using the same failed tests. If any of the executions fail, additional debugging is required to find and fix remaining bugs. This process continues until no failure is observed when the modified program is tested against the same failed tests.

Another approach is to use clustering. For example, identification of suspicious code can be divided into two steps. The first step is to group failed test cases into *fault-focusing* clusters such that those in the same cluster are related to the same fault (30). The second step is to combine failed tests in each cluster with the successful tests for debugging a single fault. Different ranking mechanisms (e.g., Tarantula (31), Crosstab (72), Coverage-based Heuristics (66), RBF neural network (70), etc.) can then be used to compute the suspiciousness of each statement and identify suspicious code that should be examined first for locating the bugs.

Podgurski et al. (53) analyze the execution profiles of failed test cases and correlate each of them with program bugs. The result is a clustering of failed executions based on the underlying bug responsible for the failure. Note that different failed tests even with respect to the same bug can have very different execution profiles. Since their clustering is based on the similarity between execution profiles, the result does not necessarily imply an accurate causation relationship between certain faults and failed executions. This limitation may degrade its fault localization capability.

Liu and Han (40) propose to cluster failed execution traces that suggest roughly the same fault location(s). They define the corresponding clustering metric R-PROXIMITY to measure the distance between failed traces. R-Proximity is computed by first using SOBER (39) to generate a predicate ranking for each test case, and then calculating the agreement between the different rankings using a weighted Kendall tau distance. Since failed traces are clustered based on the fault locations that they suggest, the results can provide clues as to the locations of the faults associated with each cluster.

Zheng et al. (83) use fault-predicting predicates to cluster failed executions. The similarity between two predicates is measured based on the product-moment correlation coefficient. First, truth probabilities for each predicate are inferred from observed data. Then, a bi-clustering algorithm uses these probabilities to jointly cluster predicates and failed executions. This is accomplished via an iterative voting process, during which each failed execution casts a vote for a predicate. The final rank of a predicate is computed based on the number of votes it has accumulated. This method is capable of clustering failed executions related to the same fault, as well as identifying predicates that may be predictors for some fault.

4.3. Impact of Test Cases

All empirical studies independent of context are sensitive to the input data. Similarly, the effectiveness of a fault localization method 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. Consider the case of (8) where Baudry et al. define a dynamic basic block as a set of statements that are all covered by the same test cases. They then use an adaptation of genetic algorithms (a so-called bacteriologic approach) to optimize a test suite and maximize the number of dynamic basic blocks. The Tarantula algorithm (31) is applied to the coverage information to rank the suspiciousness of each statement, and it is determined that the same fault localization results as Jones et al. can be achieved using fewer test cases. A related concern is that, given a fault, the same method may require more code to be examined by using one set of test cases as opposed to another.

If a program execution fails not because of the current test but because of another test which fails to set up an appropriate execution environment for the current test, then we should bundle these two test cases together as one single failed test. For example, consider two test cases t_α and t_β which are executed successively in this order. Assume also a bug in the program causes t_α to incorrectly modify values at certain memory locations but does not result in a failure of t_α. These incorrect values are to be referenced by t_β and are the only source of the failure for t_β. Under this situation we should combine t_α and t_β as a single failed test.

Budd and Angluin (13) introduce the concept of *coincidental correctness*; this refers to the circumstances under which a test case produces one or more errors in the program state, but the output of the program is still correct. Such a test case is referred to as a *coincidentally successful test*. This phenomenon can occur for many reasons; for example, if a wrong assignment is made to a variable, but the erroneous value is later overwritten, the output of the program may not have been affected. Several studies have reported that coincidental correctness can negatively impact the effectiveness of fault localization techniques. Ball et al. (7) claim that coincidental correctness resulted in the failure of their method to locate a fault in 3 out of 15 cases. Wang et al. (62) conduct a study of the Tarantula fault localization method, and they conclude that its effectiveness decreases when the frequency of coincidental correctness is high and increases when the frequency of coincidental correctness is low. Some preliminary studies such as (28) have investigated the issue, but more research is necessary to fully address the problem.

4.4. Faults introduced by missing code

One claim that can be made against the fault localization methods discussed in Section 3 is that they are incapable of locating a fault that is the result of missing code. Considering, for instance, slicing-based debugging methods, since the "faulty" code is not actually found in the program, it follows that this code will not appear in any of the slices. Based on this, one might conclude that these methods are inappropriate for locating such a fault. While this argument seems to be reasonable, it overlooks some important details. Admittedly, the missing code cannot be found in any of the slices. 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. An abnormal program execution path (and, thus, the appearance of unexpected code in the corresponding slice) with respect to a given test case should be a clue to the programmer that some omitted statements may be leading to control flow anomalies. This implies that we still should be able to identify suspicious code related to the omission error, such as the affected decision branch, using slice-based heuristics. A similar argument can also be made for program

spectrum-based (Section 3.23.2), statistics-based (Section 3.3), and program state-based methods (Section 3.4), as well as others. Although these fault localization methods cannot pinpoint the location of the missing code, they can still provide a reasonable starting point for the search.

5. CONCLUSION

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 (such as those discussed in Section 3) based on solid reasoning and supported by case studies, rather than an unsubstantiated ad hoc approach.

Some fault localization methods (e.g., (4,15,25,55,75), and others) 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 (e.g., (31,38,39,66,68,69,70,71,72), and others) 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 (31) which uses multiple failed and multiple successful tests, has been shown to be more effective than nearest neighbor (55), 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 test cases and arrive at a more detailed root-cause explanation of the failure (15,22) 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.

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.

ACKNOWLEDGMENT

The authors wish to thank Andy Restrepo of the Software Technology Advanced Research (STAR) Lab at the University of Texas at Dallas for his valuable comments in helping us preparing this paper.

REFERENCES

1. H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Debugging with Dynamic Slicing and Backtracking," *Software – Practice & Experience*, 23(6):589-616, June, 1993
2. H. Agrawal, R.A. DeMillo, and E.H. Spafford, "An Execution Backtracking Approach to Program Debugging," *IEEE Software*, 8(5):21–26, May 1991
3. H. Agrawal and J. R. Horgan, "Dynamic Program Slicing," *in Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pp. 246-256, White Plains, New York, June 1990
4. H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault Localization using Execution Slices and Dataflow Tests," *in Proceedings of the 6th IEEE International Symposium on Software Reliability Engineering*, pp. 143-151, Toulouse, France, October 1995
5. H. Agrawal, J. R. Horgan, W. E. Wong, etc., "Mining System Tests to Aid Software Maintenance," *IEEE Computer*, 31(7):64-73, July 1998
6. L. C. Ascari, L. Y. Araki, A. R. T. Pozo, and S. R. Vergilio, "Exploring Machine Learning Techniques for Fault Localization", in *Proceedings of the 10th Latin American Test Workshop*, pp. 1-6, Buzios, Brazil, March 2009.
7. T. Ball, M. Naik, and S. K. Rajamani, "From Symptom to Cause: Localizing Errors in Counterexample Traces," in *Proceedings the 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 97-105, New Orleans, Louisiana, January 2003
8. B. Baudry, F. Fleurey, and Y. Le Traon, "Improving Test Suites for Efficient Fault Localization," in *Proceedings of the 28th International Conference on Software Engineering*, pp. 82-91, Shanghai, China, May 2006
9. A. Beszedes, T. Gergely, Z. Szabo, J. Csirik, and T. Gyimothy, "Dynamic Slicing Method for Maintenance of Large C Programs," in *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*, pp. 105-113, Lisbon, Portugal, March 2001
10. F. Bourdoncle, "Abstract debugging of higher-order imperative languages," in *Proceedings of the1993 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 46-55, Albuquerque, New Mexico, USA, June 1993
11. L. C. Briand, Y. Labiche, and X. Liu, "Using Machine Learning to Support Debugging with Tarantula", in *Proceedings of the 18th IEEE International Symposium on Software Reliability*, pp. 137-146, Trollhattan, Sweden, November 2007
12. Y. Brun and M. D. Ernst, "Finding Latent Code Errors via Machine Learning over Program Executions", in *Proceedings of the 26th International Conference on Software Engineering*, pp. 480- 490, Edinburgh, UK, May 2004
13. T.A. Budd and D. Angluin, "Two Notions of Correctness and Their Relation to Testing," *Acta Infomatica*, 18(1):31-45, March 1982.
14. P. Cellier, S. Ducasse, S. Ferre, and O. Ridoux, "Formal Concept Analysis Enhances Fault Localization in Software", in *Proceedings of the 4th International Conference on Formal Concept Analysis*, pp. 273-288, Montréal, Canada, February 2008
15. H. Cleve and A. Zeller, "Locating Causes of Program Failures," in *Proceedings of the 27th International Conference on Software Engineering*, pp. 342-351, St. Louis, Missouri, USA, May, 2005
16. P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp.238-252, Los Angeles, California, USA, January 1977
17. V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight Defect Localization for Java," in *Proceedings of the 19th European Conference on Object-Oriented Programming*, pp. 528-550, Glasgow, UK, July 2005
18. R. A. DeMillo, H. Pan, and E. H. Spafford, "Critical Slicing for Software Fault Localization," in *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 121-134, San Diego, California, USA, January 1996
19. R. A. DeMillo, H. Pan, E. H. Spafford, "Failure and fault analysis for software debugging"; *in Proceedings of 21st International Computer Software and Applications Conference*, pp. 515-521, Washington DC, USA, August 1997

20. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE Transactions on Software Engineering*, 27(2):99-123, February 2001
21. L. Fausett, *Fundamentals of neural networks: architectures, algorithms, and applications*, Prentice-Hall, 1994
22. A. Groce, S. Chaki, D. Kroening, and O. Strichman, "Error Explanation with Distance Metrics," *International Journal on Software Tools for Technology Transfer*, 8(3):229-247, June 2006
23. N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating Faulty Code Using Failure-Inducing Chops," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 263-272, Long Beach, California, USA, November 2005
24. T. Gyimothy, A. Beszedes, and I. Forgacs, "An Efficient Relevant Slicing Method for Debugging," in *Proceedings of 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 303-321, Toulouse, France, September 1999
25. L. Guo, A. Roychoudhury, and T. Wang, "Accurately Choosing Execution Runs for Software Fault Localization," In *Proceedings of the 15th International Conference on Compiler Construction*, pp. 80-95**,** Vienna, Austria, March 2006
26. M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An Empirical Investigation of the Relationship between Spectra Differences and Regression Faults," *Journal of Software Testing, Verification and Reliability*, 10(3):171-194, September 2000
27. R. Hecht-Nielsen, "Theory of the back-propagation neural network," in *Proceedings of 1989 International Joint Conference on Neural Networks*, pp. 593-605, Washington DC, USA, June 1989
28. R. M. Hierons, "Avoiding Coincidental Correctness in Boundary Value Analysis," *ACM Transactions on Software Engineering and Methodology*, 15(3): 227-241, July 2006
29. http://www-static.cc.gatech.edu/aristotle/Tools/subjects
30. J. A. Jones, J. Bowring, and M. J. Harrold, "Debugging in Parallel," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pp. 16-26, London, UK, July, 2007.
31. J. A. Jones and M. J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique," in *Proceedings of the 20th IEEE/ACM Conference on Automated Software Engineering*, pp. 273-282, Long Beach, California, USA, December, 2005
32. D. Kob and F. Wotawa, "Introducing alias information into model-based debugging," in *Proceedings of the 16th Eureopean Conference on Artificial Intelligence*, pp.833-837, Valencia, Spain, August 2004
33. B. Korel, "PELAS – Program Error-Locating Assistant System," *IEEE Transactions on Software Engineering*, 14(9):1253-1260, September 1988
34. B. Korel and J. Laski, "Dynamic Program Slicing," *Information Processing Letters*, 29(3):155-163, October 1988
35. B. Korel and J. Laski, "STAD: A System for Testing and Debugging: User Perspective," in *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis*, pp.13–20, Washington DC, USA, July 1988
36. B. Korel and S. Yalamanchili, "Forward Computation of Dynamic Program Slices," in *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 66-79, Seattle, Washington, August 1994
37. C. C. Lee, P. C. Chung, J. R. Tsai, and C. I. Chang, "Robust radial basis function neural networks," *IEEE Transactions on Systems, Man, and Cybernetics: Part B Cybernetics*, 29(6):674-685, December, 1999
38. B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable Statistical Bug Isolation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 15-26, Chicago, Illinois, USA, June, 2005.
39. C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical Debugging: A Hypothesis Testing-based Approach," *IEEE Transactions on Software Engineering*, 32(10):831-848, October, 2006
40. C. Liu and J. Han, "Failure Proximity: A Fault Localization-based Approach," in *Proceedings of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 286-295, Portland, Oregon, USA, November 2006

41. C. Liu, X. Yan, H. Yu, J. Han and P. Yu, "Mining Behavior Graphs for "Backtrace" of Noncrashing Bugs," in *Proceedings of 2005 SIAM International Conference on Data Mining*, pp.286-297, Newport Beach, California, April 2005
42. C. Liu, X. Zhang, J. Han, Y. Zhang, and B.K. Bhargava, "Indexing Noncrashing Failures: A Dynamic Program Slicing-Based Approach" in *Proceedings of the 23rd International Conference on Software Maintenance*, pp. 455-464, Paris, France, October 2007.
43. J. R. Lyle and M. Weiser, "Automatic Program Bug Location by Program Slicing," in *Proceedings of the 2nd International Conference on Computer and Applications*, pp. 877-883, Beijing, China, June 1987
44. C. Mateis, M. Stumptner, D. Wieland, and F. Wotawa, "JADE - AI support for debugging Java programs," in *Proceedings of the 12th IEEE International Conference on Tools with Artificial Intelligence*, pp.62-69, Vancouver, BC, Canada, November 2002
45. C. Mateis, M. Stumptner, and F. Wotawa, "Modeling Java programs for diagnosis," in *Proceedings of the 14th European Conference on Artificial Intelligence*, pp.171-175, Berlin, Germany, August 2000
46. W. Mayer and M. Stumptner, "Modeling programs with unstructured control flow for debugging," in *Proceedings of the 15th Australian Joint Conference on Artificial Intelligence*, pp.107-118, Canberra, Australia, December 2002
47. W. Mayer and M. Stumptner, "Debugging program exceptions," in *Proceedings of the 14th International Workshop on Principles of Diagnosis*, pp.119-124, Washington, D.C., USA, June 2003
48. W. Mayer and M. Stumptner, "Approxiamte modeling for debugging of program loops," in *Proceedings of the 15th International Workshop on Principles of Diagnosis*, pp. 87-92, Carcassonne, France, June 2004
49. W. Mayer and M. Stumptner, "Abstract interpretation of programs for model-based debugging," in *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pp.471-476, Hyderabad, India, January 2007
50. W. Mayer, M. Stumptner, D. Wieland, and F. Wotawa, "Can AI help to Improve Debugging Substantially? Debugging Experiences with Value-based Models," in *Proceedings of the 15th Eureopean Conference on Artificial Intelligence*, pp. 417-421, Lyon, France, July 2002
51. W. Mayer, M. Stumptner, D. Wieland, and F. Wotawa, "Towards an Integrated Debugging Environment," in *Proceedings of the 15th Eureopean Conference on Artificial Intelligence*, pp.422-426, Lyon, France, July 2002
52. S. Nessa, M. Abedin, W. Eric Wong, L. Khan, and Y. Qi, "Fault Localization Using N-gram Analysis," in *Proceedings of the 3rd International Conference on Wireless Algorithms, Systems, and Applications*, pp. 548-559, Richardson, Texas, USA, April 2009 (Lecture Notes In Computer Science, Volume 5258)
53. A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated Support for Classifying Software Failure Reports," in *Proceedings of the 25th International Conference on Software Engineering*, pp. 465-475, Portland, Oregon, USA, May 2003
54. B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss, "Automated Fault Localization Using Potential Invariants," in *Proceedings of the 5th International Workshop on Automated and Algorithmic Debugging*, pp. 273-276, Ghent, Belgium, September 2003
55. M. Renieris and S. P. Reiss, "Fault Localization with Nearest Neighbor Queries," in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pp. 30-39, Montreal, Canada, October 2003
56. T. Reps, T. Ball, M. Das, and J. Larus, "The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem," in *Proceedings of the 6th European Software Engineering Conference*, pp. 432-449, Zurich, Switzerland, September, 1997
57. Software-artifact Infrastructure Repository at http://sir.unl.edu/portal/index.html
58. C. D. Sterling and R. A. Olsson, "Automated Bug Isolation via Program Chipping," in *Proceedings of the 6th International Symposium on Automated Analysis-Driven Debugging*, pp. 23-32, Monterey, California, USA, September 2005
59. A. B. Taha, S. M. Thebaut, and S. S. Liu, "An Approach to Software Fault Localization and Revalidation based on Incremental Data Flow Analysis," in *Proceedings of the 13th Annual International Computer Software and Applications Conference*, Washington DC, USA, pp. 527-534, September 1989

60. F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, 3(3):121–189, 1995
61. T. Wang and A. Roychoudhury, "Automated Path Generation for Software Fault Localization," in *Proceedings of 20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 347-351, Long Beach, California, USA, November 2005
62. X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang, "Taming Coincidental Correctness: Refine Code Coverage with Context Pattern to Improve Fault Localization," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 45-55, Vancouver, Canada, May 2009.
63. P. D. Wasserman, *Advanced Methods in Neural Computing*, Van Nostrand Reinhold, 1993
64. M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, SE-10(4):352-357, July 1984
65. M. Weiser, "Programmers use Slices when Debugging," *Communications of the ACM*, 25(7):446-452, July 1982
66. W. E. Wong, V. Debroy and B. Choi, "A Family of Code Coverage-based Heuristics for Effective Fault Localization," *Journal of Systems and Software*, 83(2):188-208, February, 2010
67. W. E. Wong and J. J. Li, "An Integrated Solution for Testing and Analyzing Java Applications in an Industrial Setting," in *Proceedings of the 12th IEEE Asia-Pacific Software Engineering Conference*, pp. 576-583, Taipei, Taiwan, December 2005
68. W. E. Wong and Y. Qi, "Effective Program Debugging based on Execution Slices and Inter-Block Data Dependency," *Journal of Systems and Software* , 79(7):891-903, July 2006
69. W. E. Wong and Y. Qi, "BP Neural Network-based Effective Fault Localization," *International Journal of Software Engineering and Knowledge Engineering* 19(4):573-597, June 2009
70. W. E. Wong, Y. Shi, Y. Qi, and R. Golden, "Using an RBF Neural Network to Locate Program Bugs," in *Proceedings of the 19th IEEE International Symposium on Software Reliability Engineering*, pp. 27-38, Seattle, Washington, USA, November 2008
71. W. E. Wong, T. Sugeta, Y. Qi, and J. C. Maldonado, "Smart Debugging Software Architectural Design in SDL," *Journal of Systems and Software*, 76(1):15-28, April 2005
72. W. E. Wong, T. Wei, Y. Qi, and L. Zhao, "A Crosstab-based Statistical Method for Effective Fault Localization," in *Proceedings of the 1st International Conference on Software Testing, Verification and Validation*, pp. 42-51, Lillehammer, Norway, April 2008
73. F. Wotawa, "On the relationship between model-based debugging and program slicing," *Artificial Intelligence*, 135(1-2):125-143, February 2002
74. F. Wotawa, M. Stumptner, and W. Mayer, "Model-based debugging or how to diagnose programs automatically," in *Proceedings of the 15th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems: Developments in Applied Artificial Intelligence*, pp. 746-757, Cairns, Australia, June 2002.
75. A. Zeller, "Isolating Cause-Effect Chains from Computer Programs," in *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 1-10, Charleston, South Carolina, USA, November 2002
76. A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, 28(2):183-200, February 2002
77. T. Zimmermann and A. Zeller, "Visualizing Memory Graphs," in *Proceedings of the International Seminar on Software Visualization*, pp. 191-204, Dagstuhl Castle, Germany, May 2001
78. X. Zhang, N. Gupta, and R. Gupta, "Locating Faults through Automated Predicate Switching," in *Proceedings of the 28th International Conference on Software Engineering*, pp. 272-281, Shanghai, China, May 2006
79. X. Zhang, R. Gupta, and Y. Zhang, "Precise Dynamic Slicing Algorithms," in *Proceedings of the 25th IEEE International Conference on Software Engineering*, pp. 319-329, Portland, Oregon, USA, May 2003
80. X. Zhang, R. Gupta, and Y. Zhang, "Efficient Forward Computation of Dynamic Slices Using Reduced Ordered Binary Decision Diagrams," in *Proceedings of the 26th International Conference on Software Engineering*, pp. 502-511, Edinburgh, Scotland, UK, May 2004

81. X. Zhang, H. He, N. Gupta, and R. Gupta, "Experimental Evaluation of Using Dynamic Slices for Fault Location," in *Proceedings of the 6th International Symposium on Automated Analysis-driven Debugging*, pp. 33-42, Monterey, California, USA, September 2005
82. X. Zhang, S. Tallam, N. Gupta, and R. Gupta, "Towards Locating Execution Omission Errors," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 415-424, San Diego, California, USA, June 2007.
83. A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical Debugging: Simultaneous Isolation of Multiple Bugs," in *Proceedings of the 23rd International Conference on Machine Learning*, pp. 26-29, Pittsburgh, Pennsylvania, June 2006.
84. χSuds User's Manual, Telcordia Technologies (formerly Bellcore), New Jersey, USA, 1998