International Journal of Advance Research in Science and Engineering
Vol. No.5, Issue No. 02, February 2016
www.ijarse.com
IJARSE
ISSN 2319 - 8354

A STUDY OF SOFTWARE FAULT LOCALIZATION TECHNIQUES

Anurag Negi[1], R.P. Mahapatra[2]

[1]*Mtech Student,Department of ComputerScience and Engineering,SRM University (India)*

[2]*Head of Department, Department of ComputerScience 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 demandfor automatic fault localization techniques that can guide programmers to thelocations of faults, with minimal human intervention. This demand has led to theproposal and development of various methods, each of which seeks to make thefault localization process more effective in its own unique and creative way. In thisarticle we provide an overview of several such methods and discuss some of thekey 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 thatsoftware 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 aspossible without introducing new bugs at the same time.

During program debugging, fault localization is the activity of identifying the exact locations ofprogram faults. It is a very expensive and time consuming process. Its effectiveness depends ondevelopers' understanding of the program being debugged, their ability of logical judgment, pastexperience in program debugging, and how suspicious code, in terms of its likelihood ofcontaining 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 facilitatefault localization and make it more effective. While these methods share similar goals, they canbe quite different from one another and often stem from ideas that themselves originate fromseveral 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 theirexperience, 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 provide 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 itsinter-block data dependency with the code which is currently being examined, whereas the latterexcludes less suspicious code from the search domain using the execution slices of additionalsuccessful tests. Different execution slice-based debugging tools have been developed and used inpractice 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 asexecution information for conditional branches or loop-free intra-procedural paths (8). It can beused to track program behavior (9). When the execution fails, such information can be used toidentify suspicious code that is responsible for the failure. Early studies (10,11,12,13) only usefailed test cases for fault localization, though this approach has subsequently been deemedineffective (14,15,16). These later studies achieve better results using both the successful andfailed test cases and emphasizing the contrast between them.

The Executable Statement Hit Spectrum (ESHS) records which executable statements areexecuted. Two ESHS-based fault localization methods, set union and set intersection, areproposed in (17). The set union computes the set difference between the program spectra of afailed test and the union spectra of a set of successful tests. It focuses on the source code that isexecuted by the failed test but not by any of the successful tests. Such code is more suspiciousthan others. The set intersection method excludes the code that is executed by all the successfultests 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 interms of the “distance” between them. In their method, the execution of a test is represented as

asequence of statements that are sorted by their execution counts. If a bug is in the difference set, itis located. For a bug that is not contained in the difference set, the method can continue the buglocalization by first constructing a program dependence graph and then including and checkingadjacent 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 coverageand 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 = (numberof successful tests that execute the statement)/(total number of successful tests). One problemwith Tarantula is that it does not distinguish the contribution of one failed test case from anotheror one successful test case from another.In (16), Wong et al. address two important issues: first, how can each additional failed test caseaid in locating program bugs; and second, how can each additional successful test case help inlocating program bugs. They propose that with respect to a piece of code, the contribution of thefirst failed test case that executes it in computing its likelihood of containing a bug is larger thanor equal to that of the second failed test case that executes it, which in turn is larger than or equalto that of the third failed test case that executes it, and so on. This principle is also applied to thecontribution 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 programbug, by examining less code before the first faulty statement containing the bug is identified, thanother fault localization methods such as set union, set intersection, nearest neighbor (17) andcause 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 tobe compared to a successful run, then which successful run should it be compared to? They do soby proposing a control flow-based difference metric that takes into account the sequence ofstatement executions in two runs instead of just the set of statement executions. Given a failed runand a pool of successful runs, they choose that successful run whose execution sequence is closestto the failed run based on the difference metric. Then, a bug report is generated by returning thedifference 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 tothe failed test (in terms of their execution slices) in order to filter out as much code as possible. Inthis way, we start the fault localization with a very small set of suspicious code, and then increasethe search domain, if necessary, using an inter-block data dependency-based augmentationmethod.

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

• Predicate Count Spectrum (PRCS)-based: PRCS records how predicates are executed. Suchinformation can be used to track program behaviors that are likely to be erroneous. Thesemethods are often referred to as *statistical debugging* because the PRCS information is

analyzed using statistical methods. Fault localization methods in this category includeLiblit05 (21), SOBER (22), etc. See “Statistics-based Methods” for more details.

• Program Invariants Hit Spectrum (PIHS)-based: This spectrum records the coverage ofprogram invariants (23), which are the program properties that should be preserved inprogram executions. PIHS-based methods try to find violations of program properties infailed program executions to locate bugs. A study on the fault localization

using "potentialinvariants" is reported by Pytlik, et al. (24). The major obstacle in applying such methods ishow 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 ofsuccessful executions as the program properties.

• Method Calls Sequence Hit Spectrum (MCSHS)-based: Information is collected regardingthe sequences of method calls covered during program execution. For the purposes of faultlocalization, this data is helpful when applied to object-oriented software. In some cases, sucha program may not fail even if the faulty code is executed; a particular sequence of methodcalls 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 theidentification and analysis of method call sequences. Both incoming method calls (how anobject 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 thecalling and transition relationships between functions. They define a framework to mineclosed frequent graphs from these behavior graphs and use them as a training set forclassifiers that will identify suspicious functions.

C. Statistics-based Methods: Liblit et al. propose a statistical debugging algorithm (referred to as Liblit05) that can isolate bugsin the programs with instrumented predicates at particular points (21). Feedback reports aregenerated 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 thatthe execution of P implies failure. Predicates that have $Failure(P) - Context(P) \leq 0$ are discarded.

Remaining predicates are prioritized based on their "*importance*" scores, which gives anindication of the relationship between predicates and program bugs. Predicates with a higherscore should be examined first to help programmers find bugs. Once a bug is found and fixed, thefeedback reports related to that particular bug are removed. This process continues to find otherbugs 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 beevaluated to be true more than once in a run. Compute $\pi(P)$ which is the probability that P isevaluated to be true in each run as $\pi(P) = n(f)/(n(t) + n(f))$ where $n(t)$ is the number of times P is evaluatedto be true in a specific run and $n(f)$ is the number of times P is evaluated as false. If thedistribution 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 ofits 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 categoricalvariables "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 eachstatement. However, the exact suspiciousness of each statement depends on the degree ofassociation (instead of the result of the hypothesis testing) between its coverage (number of teststhat 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. Itcan be a good indicator for locating program bugs. A general approach for using program statesin fault localization is to modify the values of some variables to determine which one is the causeof erroneous program execution.

Zeller, et al. propose a program state-based debugging approach, delta debugging (30,31), toreduce the causes of failures to a small set of variables by contrasting program states betweenexecutions of a successful test and a failed test via their memory graphs (32). Variables are testedfor suspiciousness by replacing their values from a successful test with their corresponding valuesfrom the same point in a failed test and repeating the program execution. Unless the identicalfailure is observed, the variable is no longer considered suspicious.

Delta debugging is extended to the cause transition method by Cleve and Zeller (18) to identifythe locations and times where the cause of failure changes from one variable to another. Analgorithm named *cts*is proposed to quickly locate cause transitions in a program execution. Apotential problem of the cause transition method is that the cost is relatively high; there may existthousands of states in a program execution, and delta debugging at each matching point requiresadditional test runs to narrow down the causes. Another problem is that the identified locationsmay not be the place where the bugs reside. Gupta et al. (34) introduce the concept of failureinducingchop as an extension to the cause transition method to overcome this issue. First, deltadebugging is used to identify input and output variables that are causes of failure. Dynamic slicesare then computed for these variables, and the code at the intersection of the forward slicing ofthe 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 faultlocalization method where program states are changed to forcefully alter the executed branches ina failed execution. A predicate whose switch can make the program execute successfully islabeled 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 andPrioritization-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 ofa failed test and alters the outcome of branches in that path to produce a successful execution. Thebranch 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 throughexperience. 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 manydisciplines 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 faultbased on input data such as statement coverage, etc. It should therefore come as no surprise thatthe 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) neuralnetwork which is one of the most popular neural network models in practice (38). A BP neuralnetwork has a simple structure, which makes it easy to implement using computer programs. Atthe 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 whichstatements are executed by which test case) and the corresponding execution result (success orfailure) are collected. Together, they are used to train a BP neural network so that the network canlearn the relationship between them. Then, the coverage of a set of virtual test cases that eachcovers only one statement in the program are input to the trained BP network, and the outputs canbe 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 lesssusceptible to these problems and have a faster learning rate (41,42). The RBF network issimilarly trained against the coverage data and execution results collected for each test case, andthe 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 mightclassify test cases into various partitions such that failed test cases in the same partition mostlikely fail due to the same fault. The underlying premise is that distinct failure conditions for testcases can be identified depending on the inputs and outputs of the test case (categorypartitioning). 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 usedto rank the statements using a heuristic similar to Tarantula (15) to form a ranking based on eachpartition. These individual rankings are then consolidated to form a final statement ranking whichcan then be examined to locate the faults.

Brun and Ernst (28) build a learning model using machine learning (e.g., Support VectorMachines) to distinguish faulty and non-faulty programs using static analysis. General programproperties (e.g., variables that are not initialized) are assumed to likely indicate the faults inprograms and therefore in the learning model, properties of correct and incorrect programs areused to build the model. The classification step involves feeding as input the properties of a newprogram, and then the properties are ranked according to the strength of their association withfaulty 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) forfault localization.

G.Model-Based Methods: For model-based methods, the model used in each method is an important topic of researchbecause the expressive capability of each model is crucial to the effectiveness of that method inlocating 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 ofprogram failures and to analyze the nature of program failures, respectively. Failure modes areused to answer “How do we know the execution of a program fails?” and failure types are used toanswer “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-establishedrelationships between failure modes and failure types, certain failure types can be identified aspossible causes for the failure. Heuristics based on dynamic instrumentation (such as dynamicprogram slice) and testing information are then used to reduce the search domain for localizingthe fault by predicting possible

faulty statements. One significant problem of using this model isthat it is extremely difficult, if not impossible, to obtain an exhaustive list of failure modesbecause different programs can have very different abnormal behavior and symptoms when theyfail. As a result, we do not have a complete relationship between all possible failure modes andfailure types. This implies we might not be able to identify possible failure types responsible forthe failure being analyzed.

Wotawa, et al.(47) propose to construct dependency models based on a source code analysis ofthe 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. Ifthe execution of the target program on a test case fails, conflicts between the test case and themodels will be determined to find fault candidates. For each statement, a default assumption ismade to suggest whether the statement is correct or incorrect. These assumptions will be revisedduring fault localization until the failure can be explained. The limitation is that their study onlyfocuses 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, recursiveprocedures, 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 usingrelevant information extracted from data. Data mining can uncover hidden patterns in samples ofdata (which have been *mined*) that may not, and often will not, be discovered by manual analysisalone. Also sometimes the sheer volume of data that is available for analysis far exceeds thatwhich can be analyzed by humans alone. Efficient data mining techniques transcend suchproblems 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. Inaddition, although the *complete* execution trace (including the actual order of execution of eachstatement) of a program collected during the testing phase is a valuable resource for finding thelocation 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 traditionallydeal 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 thetrace data. The failed execution traces are then examined to find the *N*-grams with a rate ofoccurrence higher than a certain threshold in the failed executions. A statistical analysis isconducted to determine the conditional probability that an execution fails given that a certain *N*gramappears in its trace – this probability is known as the "confidence" for that *N*-gram. *N*-gramsare sorted by descending order of confidence and the corresponding statements in the program aredisplayed based on their first appearance in the list. Case studies which apply this have shown that it achieves fault localizationmore effectively than Tarantula (15), by requiring the examination of less code before the firstfaulty 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 statementexecution 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 thatshould be covered by a rule to be selected. A

large number of rules so generated, can be partiallyordered 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 notnecessarily applicable for every program. Choosing an effective debugging strategy normallyrequires expert knowledge regarding the program in question. In general, an experiencedprogrammer's intuition about the location of the bug should be explored first. However, if thisfails, an appropriate fallback would be a systematic fault localization method based on solid reasoning and supported by case studies, rather than anunsubstantiated ad hoc approach.

Some fault localization methods are restricted to selecting onlya single failed test case and a single successful test case, based on certain criteria, to locate a bug.Alternative methods rely on the combined datafrom 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 aprogram bug, in that they require the programmer to examine less code before the first faultylocation is discovered. For example, the Tarantula method (15) which uses multiple failed andmultiple successful tests, has been shown to be more effective than nearest neighbor (17), amethod that only uses a single failed and single successful test. However, it is important to notethat 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 themethods that take into account multiple successful and failed test cases simultaneously. Neithercategory is necessarily superior to the other, but a general rule is that an effective faultlocalization method should assign higher suspiciousness to code that is likely to contain bugs andlower suspiciousness to code in which the presence of bugs is less probable. This increases thelikelihood that the fault will appear near the top of the list when the code is prioritized forexamination based on suspiciousness. An effective fault localization method should also,whenever possible, assign a unique suspiciousness value to each unit of code to reduce ambiguityduring 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 cam 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, faultlocalization is far from perfect. While these methods are constantly advancing, software too isbecoming increasingly more complex which means the challenges posed by fault localization arealso growing. Thus, there is a significant amount of research still to be done, and a large numberof breakthroughs yet to be made.

REFERENCES

[1] F. Tip, "A survey of program slicing techniques," Journal of Programming Languages, 3(3):121–189, 1995

[2] M. Weiser, "Programmers use Slices when debugging," Communications of the ACM, 25(7):446-452, July 1982

[3] M. Weiser, "Program slicing," IEEE Transactions on Software Engineering, SE-10(4):352- 357, July 1984

[4] 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

[5] H. Agrawal, J. R. Horgan, W. E. Wong, etc., "Mining System Tests to Aid SoftwareMaintenance," IEEE Computer, 31(7):64-73, July 1998

[6]χSuds User's Manual, Telcordia Technologies (formerly Bellcore), New Jersey, USA, 1998

[7]W. E. Wong and J. J. Li, "An Integrated Solution for Testing and Analyzing JavaApplications in an Industrial Setting," in Proceedings of the 12th IEEE Asia-Pacific SoftwareEngineering Conference, pp. 576-583, Taipei, Taiwan, December 2005

[8]. M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An Empirical Investigation of theRelationship between Spectra Differences and Regression Faults," Journal of SoftwareTesting, Verification and Reliability, 10(3):171-194, September 2000

[9]. T. Reps, T. Ball, M. Das, and J. Larus, "The Use of Program Profiling for SoftwareMaintenance with Applications to the Year 2000 Problem," in Proceedings of the 6thEuropean Software Engineering Conference, pp. 432-449, Zurich, Switzerland, September,1997

[10]. H. Agrawal, R.A. DeMillo, and E.H. Spafford, "An Execution Backtracking Approach toProgram Debugging," IEEE Software, 8(5):21–26, May 1991

[11]. B. Korel, "PELAS – Program Error-Locating Assistant System," IEEE Transactions onSoftware Engineering, 14(9):1253-1260, September 1988

[12]. B. Korel and J. Laski, "STAD: A System for Testing and Debugging: User Perspective," inProceedings of the 2nd Workshop on Software Testing, Verification, and Analysis, pp.13–20,Washington DC, USA, July 1988

[13]. A. B. Taha, S. M. Thebaut, and S. S. Liu, "An Approach to Software Fault Localization andRevalidation based on Incremental Data Flow Analysis," in Proceedings of the 13th AnnualInternational Computer Software and Applications Conference, Washington DC, USA, pp. 527-534, September 1989

[14]. H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault Localization using ExecutionSlices and Dataflow Tests," in Proceedings of the 6th IEEE International Symposium onSoftware Reliability Engineering, pp. 143-151, Toulouse, France, October 1995

[15]. 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 AutomatedSoftware Engineering, pp. 273-282, Long Beach, California, USA, December, 2005

[16]. W. E. Wong, V. Debroy and B. Choi, "A Family of Code Coverage-based Heuristics forEffective Fault Localization," Journal of Systems and Software, 83(2):188-208, February,2010

[17]. M. Renieris and S. P. Reiss, "Fault Localization with Nearest Neighbor Queries," inProceedings of the 18th IEEE International Conference on Automated Software Engineering,pp. 30-39, Montreal, Canada, October 2003

[18]. H. Cleve and A. Zeller, "Locating Causes of Program Failures," in Proceedings of the 27thInternational Conference on Software Engineering, pp. 342-351, St. Louis, Missouri, USA,May, 2005

[19]. L. Guo, A. Roychoudhury, and T. Wang, "Accurately Choosing Execution Runs for SoftwareFault Localization," In Proceedings of the 15th International Conference on CompilerConstruction, pp. 80-95**,** Vienna, Austria, March 2006

[20]. "A New Optimized GA-RBF Neural Network Algorithm" WeikuanJia, 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

[21]. B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable Statistical BugIsolation," in Proceedings of the 2005 ACM SIGPLAN Conference on ProgrammingLanguage Design and Implementation, pp. 15-26, Chicago, Illinois, USA, June, 2005.

[22]. C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical Debugging: A HypothesisTesting-based Approach," IEEE Transactions on Software Engineering, 32(10):831-848,October, 2006

[23]. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically Discovering LikelyProgram Invariants to Support Program Evolution," IEEE Transactions on SoftwareEngineering, 27(2):99-123, February 2001

[24]. B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss, "Automated Fault LocalizationUsing Potential Invariants," in Proceedings of the 5th International Workshop on Automatedand Algorithmic Debugging, pp. 273-276, Ghent, Belgium, September 2003

[25]. V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight Defect Localization for Java," inProceedings of the 19th European Conference on Object-Oriented Programming, pp. 528-550, Glasgow, UK, July 2005

[26]. C. Liu, X. Yan, H. Yu, J. Han and P. Yu, "Mining Behavior Graphs for "Backtrace" ofNon-crashing Bugs," in Proceedings of 2005 SIAM International Conference on Data Mining,pp.286-297, Newport Beach, California, April 2005

[27]. 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 ACMSIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp.238-252, LosAngeles, California, USA, January 1977

[28]. Y. Brun and M. D. Ernst, "Finding Latent Code Errors via Machine Learning over ProgramExecutions", in Proceedings of the 26th International Conference on Software Engineering,
pp. 480- 490, Edinburgh, UK, May 2004

[29]. W. E. Wong, T. Sugeta, Y. Qi, and J. C. Maldonado, "Smart Debugging SoftwareArchitectural Design in SDL," Journal of Systems and Software, 76(1):15-28, April 2005

[30]. A. Zeller, "Isolating Cause-Effect Chains from Computer Programs," in Proceedings of the10th ACM SIGSOFT Symposium on Foundations of Software Engineering, pp. 1-10,Charleston, South Carolina, USA, November 2002

[31]. A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," IEEETransactions on Software Engineering, 28(2):183-200, February 2002 [30]

[32]. T. Zimmermann and A. Zeller, "Visualizing Memory Graphs," in Proceedings of theInternational Seminar on Software Visualization, pp. 191-204, Dagstuhl Castle, Germany,May 2001

[33] S. Nessa, M. Abedin, W. Eric Wong, L. Khan, and Y. Qi, "Fault Localization Using N-gramAnalysis," in Proceedings of the 3rd International Conference on Wireless Algorithms,Systems, and Applications, pp. 548-559, Richardson, Texas, USA, April 2009 (Lecture NotesIn Computer Science, Volume 5258)

[34] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating Faulty Code Using Failure-InducingChops," in Proceedings of the 20th IEEE/ACM International Conference on AutomatedSoftware Engineering, pp. 263-272, Long Beach, California, USA, November 2005

[35] X. Zhang, N. Gupta, and R. Gupta, "Locating Faults through Automated PredicateSwitching," in Proceedings of the 28th International Conference on Software Engineering,pp. 272-281, Shanghai, China, May 2006

[36] T. Wang and A. Roychoudhury, "Automated Path Generation for Software FaultLocalization," in Proceedings of 20th IEEE/ACM International Conference on AutomatedSoftware Engineering, pp. 347-351, Long Beach, California, USA, November 2005

[37] 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

[38] L. Fausett, Fundamentals of neural networks: architectures, algorithms, and applications,Prentice-Hall, 1994

[39] R. Hecht-Nielsen, "Theory of the back-propagation neural network," in Proceedings of 1989International Joint Conference on Neural Networks, pp. 593-605, Washington DC, USA,June 1989

[40] W. E. Wong, Y. Shi, Y. Qi, and R. Golden, "Using an RBF Neural Network to LocateProgram Bugs," in Proceedings of the 19th IEEE International Symposium on SoftwareReliability Engineering, pp. 27-38, Seattle, Washington, USA, November 2008

[41] C. C. Lee, P. C. Chung, J. R. Tsai, and C. I. Chang, "Robust radial basis function neuralnetworks," IEEE Transactions on Systems, Man, and Cybernetics: Part B Cybernetics,29(6):674-685, December, 1999

[42] P. D. Wasserman, Advanced Methods in Neural Computing, Van Nostrand Reinhold, 1993

[43] L. C. Briand, Y. Labiche, and X. Liu, "Using Machine Learning to Support Debugging withTarantula", in Proceedings of the 18th IEEE International Symposium on Software Reliability,pp. 137-146, Trollhattan, Sweden, November 2007

[44] L. C. Ascari, L. Y. Araki, A. R. T. Pozo, and S. R. Vergilio, "Exploring Machine LearningTechniques for Fault Localization", in Proceedings of the 10th Latin American Test Workshop, pp. 1-6, Buzios, Brazil, March 2009.

[45] P. Cellier, S. Ducasse, S. Ferre, and O. Ridoux, "Formal Concept Analysis Enhances FaultLocalization in Software", in Proceedings of the 4th International Conference on FormalConcept Analysis, pp. 273-288, Montréal, Canada, February 2008

[46] 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

[47] F. Wotawa, M. Stumptner, and W. Mayer, "Model-based debugging or how to diagnoseprograms automatically," in Proceedings of the 15th International Conference on Industrialand Engineering Applications of Artificial Intelligence and Expert Systems: Developments inApplied Artificial Intelligence, pp. 746-757, Cairns, Australia, June 2002.

[48] W. Mayer and M. Stumptner, "Approxiamte modeling for debugging of program loops," inProceedings of the 15th International Workshop on Principles of Diagnosis, pp. 87-92,Carcassonne, France, June 2004

[49] F. Bourdoncle, "Abstract debugging of higher-order imperative languages," in Proceedings ofthe1993 ACM SIGPLAN Conference on Programming Language Design andImplementation, pp. 46-55, Albuquerque, New Mexico, USA, June 1993