A Survey on Software Fault Localization

Article *in* IEEE Transactions on Software Engineering · August 2016

DOI: 10.1109/TSE.2016.2521368

A Survey on Software Fault Localization

W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa, *Member, IEEE*

Abstract—Software fault localization, the act of identifying the locations of faults in a program, is widely recognized to be one of the most tedious, time consuming, and expensive – yet equally critical – activities in program debugging. Due to the increasing scale and complexity of software today, manually locating faults when failures occur is rapidly becoming infeasible, and consequently, there is a strong demand for techniques that can guide software developers to the locations of faults in a program with minimal human intervention. This demand in turn has fueled the proposal and development of a broad spectrum of fault localization techniques, each of which aims to streamline the fault localization process and make it more effective by attacking the problem in a unique way. In this article, we catalog and provide a comprehensive overview of such techniques and discuss key issues and concerns that are pertinent to software fault localization as a whole.

Index Terms—Software fault localization, program debugging, software testing, execution trace, suspicious code, survey

1 Introduction

Software is fundamental to our lives today, and with its ever-increasing usage and adoption, its influence is practically ubiquitous. In fact, at present, software is not just employed in, but is critical to, many security and safety-critical systems in industries such as medicine, aeronautics, and nuclear energy. Not surprisingly, this trend has been accompanied by a drastic increase in the scale and complexity of software. Unfortunately, this has also resulted in more software bugs, which often lead to execution failures with huge losses [260], [275], [365]. Furthermore, software faults in safety-critical systems have significant ramifications, including not only financial loss, but also potential loss of life, which is an alarming prospect [368]. A 2002 report from the National Institute of Standards and Technology (NIST) [304] indicated that software errors are estimated to cost the U.S. economy $59.5 billion annually (0.6 percent of the GDP); the cost has undoubtedly grown since then. Over half the cost of fixing or responding to these bugs is passed on to software users, while software developers and vendors absorb the rest.

Even when faults in software are discovered due to erroneous behavior or some other manifestation of the fault(s),[1] finding and fixing them is an entirely different matter. Fault localization, which focuses on the former, i.e., identifying the locations of faults, has historically been a manual task that has been recognized to be time consuming and tedious as well as prohibitively expensive [347], given the size and complexity of large-scale software systems today. Furthermore, manual fault localization relies heavily on the software developer's experience, judgment, and intuition to identify and prioritize code that is likely to be faulty. These limitations have led to a surge of interest in developing techniques that can partially or fully automate the localization of faults in software while reducing human input. Though some techniques are similar and some very different (in terms of the type of data consumed, the program components focused on, comparative effectiveness and efficiency, etc.), they each try to attack the problem of fault localization from a unique perspective, and typically offer both advantages and disadvantages relative to one another. With many techniques already in existence and others continually being proposed, as well as with advances being made both from a theoretical and practical perspective, it is important to catalog and overview current techniques in fault localization in order to offer a comprehensive resource for those already in the area as well as those interested in making contributions to it.

In order to provide a complete survey covering most of the publications related to software fault localization since the late 1970s, we created a publication repository that includes 331 papers published from 1977 to November 2014. We also searched for Masters' and Ph.D. theses closely related to software fault localization, which are listed in Table 1.

All papers in our repository[2] are sorted by year, and the result is displayed in Fig. 1. As shown in the figure, the number of publications grew rapidly after 2001, indicating that more and more researchers began to devote themselves to the area of software fault localization over the last 10 years.

Also, as per our repository, Fig. 2 gives the number of publications related to software fault localization that have

[1] In this survey the terms 'software' and 'program' are used interchangeably. Also 'fault' and 'bug' are used interchangeably.

[2] In the rest of paper, "all papers" is used to represent "all papers in our repository."

- *W.E. Wong is with the State Key Laboratory of Software Engineering, Wuhan University, China, and the Department of Computer Science, University of Texas at Dallas, Richardson, TX. E-mail: ewong@utdallas.edu.*
- *R. Gao and Y. Li are with the Department of Computer Science, University of Texas at Dallas, Richardson, TX.*
  E-mail: {gxr116020, yxl107221}@utdallas.edu.
- *R. Abreu is with the Department of Informatics Engineering, University of Porto, Portugal and the Palo Alto Research Center (PARC), Palo Alto, CA. E-mail: rui@computer.org.*
- *F. Wotawa is with the Institute for Software Technology, Graz University of Technology, Austria. E-mail: wotawa@ist.tugraz.at.*

Manuscript received 19 Feb. 2015; revised 15 Dec. 2015; accepted 11 Jan. 2016. Date of publication 24 Jan. 2016; date of current version 19 Aug. 2016.
Recommended for acceptance by A. Zeller.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2016.2521368

TABLE 1
A List of Recent Ph.D. and Master's Theses on Software Fault Localization

Author	Tittle	Degree	University	Year
Ehud Y. Shapiro [325]	Algorithmic Program Debugging	Ph.D.	Yale University	1983
Hiralal Agrawal [17]	Towards Automatic Debugging of Computer Programs	Ph.D.	Purdue University	1991
Hsin Pan [276]	Software debugging with dynamic instrumentation and test-based knowledge	Ph.D.	Purdue University	1993
W. Bond. Gregory [131]	Logic Programs for Consistency-based Diagnosis	Ph.D.	Carleton University	1994
Benjamin Robert Liblit [221]	Cooperative Bug Isolation	Ph.D.	The University of California, Berkeley	2004
Bernhard Peichl [289]	Automated Source-Level Debugging of Synthesizeable VHDL Designs	Ph.D.	Graz University of Technology	2004
Haifeng He [153]	Automated Debugging using Path-based Weakest Preconditions	Master	University of Arizona	2004
Alex David Groce [135]	Error Explanation and Fault Localization with Distance Metrics	Ph.D.	Carnegie Mellon University	2005
Emmanuel Renieris [302]	A Research Framework for Software-Fault Localization Tools	Ph.D.	Brown University	2005
Daniel Köb [197]	Extended Modeling for Automatic Fault Localization in Object-Oriented Software	Ph.D.	Graz University of Technology	2005
David Hovemeyer [166]	Simple and Effective Static Analysis to Find Bugs	Ph.D.	University of Maryland	2005
Peifeng Hu [167]	Automated Fault Localization: A Statistical Predicate Analysis Approach	Ph.D.	The University of Hong Kong	2006
Xiangyu Zhang [412]	Fault Localization via Precise Dynamic Slicing	Ph.D.	The University of Arizona	2006
Rafi Vayani [346]	Improving Automatic Software Fault Localization	Master	Delft University of Technology	2007
Ramana Rao Kompella [199]	Fault Localization in Backbone Networks	Ph.D.	University of California, San Diego	2007
Andreas Griesmayer [132]	Debugging Software: From Verification to Repair	Ph.D.	Graz University of Technology	2007
Tao Wang [351]	Post-Mortem Dynamic Analysis For Software Debugging	Ph.D.	Fudan University	2007
Sriraman Tallam [342]	Fault Location and Avoidance in Long-Running Multithreaded Applications	Ph.D.	The University of Arizona	2007
Ophelia C. Chesley [73]	CRISP-A fault localization Tool for Java Programs	Master	Rutgers, The State University of New Jersey	2007
Shan Lu [229]	Understanding, Detecting and Exposing Concurrency Bugs	Ph.D.	University of Illinois at Urbana-Champaign	2008
Naveed Riaz [306]	Automated Source-Level Debugging of Synthesizable Verilog Designs	Ph.D.	Graz University of Technology	2008
James Arthur Jones [183]	Semi-Automatic Fault Localization	Ph.D.	Georgia Institute of Technology	2008
Zhenyu Zhang [413]	Software Debugging through Dynamic Analysis of Program Structures	Ph.D.	The University of Hong Kong	2009
Rui Abreu [5]	Spectrum-based Fault Localization in Embedded Software	Ph.D.	Delft University of Technology	2009
Dennis Jefferey [171]	Dynamic State Alteration Techniques for Automatically Locating Software Errors	Ph.D.	University of California Riverside	2009
Xinming Wang [354]	Automatic Localization of Code Omission Faults	Ph.D.	Hong Kong University of Science and Technology	2010
Fabrizio Pastore [286]	Automatic Diagnosis of Software Functional Faults by Means of Inferred Behavioral Models	Ph.D.	University of Milan Bicocca	2010
Mihai Nica [270]	On the Use of Constraints in Automated Program Debugging –From Foundations to Empirical Results	Ph.D.	Graz University of Technology	2010
Zachary P.Fry [118]	Fault Localization Using Textual Similarities	Master	The University of Virginia	2011
Hua Jie Lee [182]	Software Debugging Using Program Spectra	Ph.D.	The University of Melbourne	2011
Vidroha Debroy [89]	Towards the Automation of Program Debugging	Ph.D.	The University of Texas at Dallas	2011
Alberto Gonzalez Sanchez [318]	Cost Optimizations in Runtime Testing and Diagnosis	Ph.D.	Delft University of Technology	2011
Jared David DeMott [98]	Enhancing Automated Fault Discovery and Analysis	Ph.D.	Michigan State University	2012
Xin Zhang [404]	Secure and Efficient Network Fault Localization	Ph.D.	Carnegie Mellon University	2012
Xiaoyuan Xie [383]	On the Analysis of Spectrum-based Fault Localization	Ph.D.	Swinburne University of Technology	2012
Alexandre Perez [292]	Dynamic Code Coverage with Progressive Detail Levels	Master	University of Porto	2012
Raul Santelices [319]	Change-effects Analysis for Effective Testing and Validation of Evolving Software	Ph.D.	Georgia Institute of Technology	2012
George. K. Baah [41]	Statistical Causal Analysis for Fault Localization	Ph.D.	Georgia Institute of Technology	2012
Swarup K. Sahoo [316]	A Novel Invariants-based Approach for Automated Software Fault Localization	Ph.D.	University of Illinois at Urbana-Champaign	2012
Birgit Hofer [163]	From Fault Localization of Programs written in 3rd level Language to Spreadsheets	Ph.D.	Graz University of Technology	2013
Aritra Bandyopadhyay [47]	Mitigating the Effect of Coincidental Correctness in Spectrum based Fault Localization	Ph.D.	Colorado State University	2013
Shounak Roychowdhury [312]	A Mixed Approach to Spectrum-based Fault Localization Using Information Theoretic Foundations	Ph.D.	The University of Texas at Austin	2013
Shaimaa Ali [26]	Localizing State-Dependent Faults Using Associated Sequence Mining	Ph.D.	The University of Western Ontario	2013
Christian Kuhnert [206]	Data-driven Methods for Fault Localization in Process Technology	Ph.D.	Karlsruhe Institute of Technology	2013
Dawei Qi [295]	Semantic Analyses to Detect and Localize Software Regression Errors	Ph.D.	Tsinghua University	2013
William N. Sumner [337]	Automated Failure Explanation Through Execution Comparison	Ph.D.	Purdue University	2013
Mark A. Hays [151]	A Fault-based Model of Fault Localization Techniques	Ph.D.	University of Kentucky	2014
Sang Min Park [284]	Effective Fault Localization Techniques for Concurrent Software	Ph.D.	Georgia Institute of Technology	2014
Gang Shu [327]	Statistical Estimation of Software Reliability and Failure-causing Effect	Ph.D.	Case Western Reserve University	2014
Lucia [233]	Ranking-based Approaches for Localizing Faults	Ph.D.	Singapore Management University	2014
Seok-Hyeon Moon [259]	Effective Software Fault Localization using Dynamic Program Behaviors	Master	Korea Advanced Institute of Science and Technology	2014
Yepang Liu [228]	Automated Analysis of Energy Efficiency and Performance for Mobile Applications	Ph.D.	The Hong Kong University of Science and Technology	2014
Cuiting Chen [68]	Automated Fault Localization for Service-Oriented Software Systems	Ph.D.	Delft University of Technology	2015
Matthias Rohr [308]	Workload-sensitive Timing Behavior Analysis for Fault Localization in Software Systems	Ph.D.	Kiel University	2015

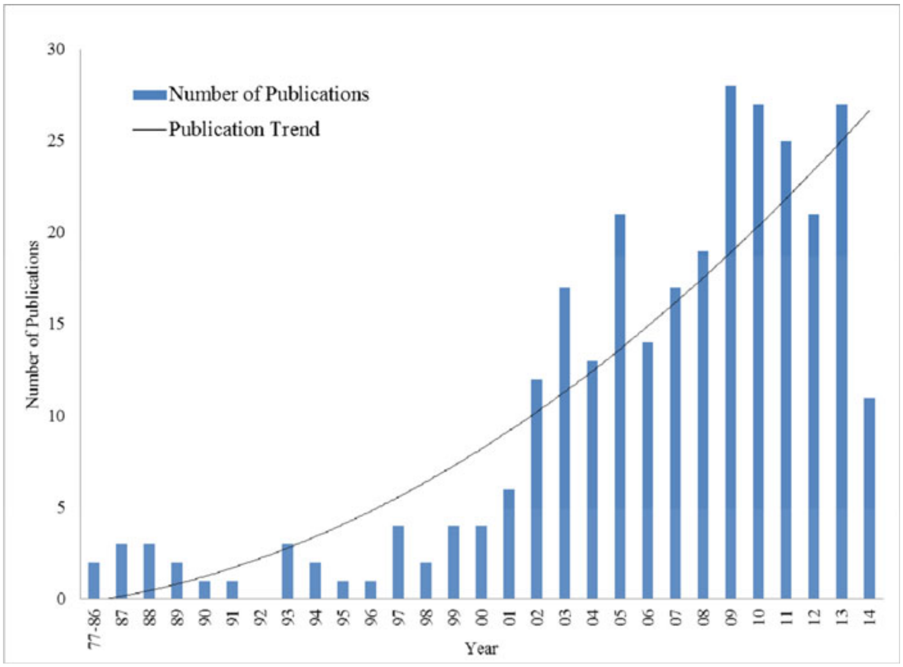


Fig. 1. Papers on software fault localization from 1977 to November 2014.

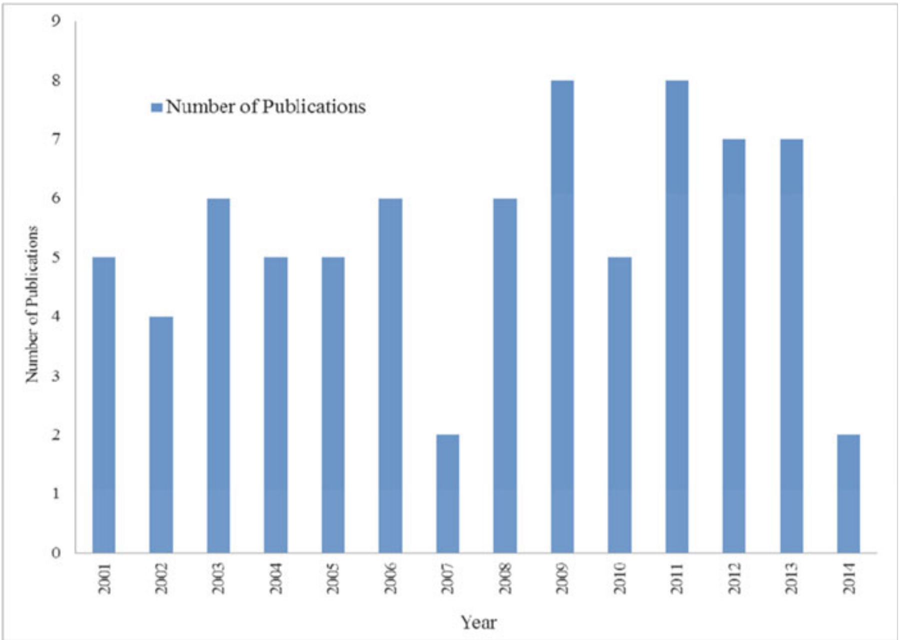


Fig. 2. Publication of software fault localization in top venues from 2001 to November 2014.

appeared in top quality and leading journals and conferences that focus on Software Engineering – IEEE Transactions on Software Engineering, ACM Transactions on Software Engineering and Methodology, International Conference on Software Engineering, ACM International Symposium on Foundations of Software Engineering, and ACM International Conference on Automated Software Engineering – from 2001 to November 2014. This trend again supports the claim that software fault localization is not just an important but also a popular research topic and has been discussed very heavily in top quality software engineering journals and conferences over the last ten years.

There is thus a rich collection of literature on various techniques that aim to facilitate fault localization and make it more effective.[3] Despite the fact that these techniques share similar goals, they can be quite different from one another and often stem from ideas that originate from several different disciplines. While we aim to comprehensively cover as many fault localization techniques as possible, no article, regardless of breadth or depth, can cover all of them. In this survey, our primary focus is on the techniques for

[3] Refer to Section 5 for more details on how to evaluate the effectiveness of a software fault localization technique.

locating Bohrbugs [139]. Those for diagnosing Mandelbugs [139] such as performance bugs, memory leaks, software bloats, and security vulnerabilities are not included in the scope. Also, due to space limitations, we group techniques into appropriate categories for collective discussion with an emphasis on the most important features and leave other details of these techniques to their respectively published papers. This is especially the case for techniques targeting a specific application domain, such as fault localization for concurrency bugs and spreadsheets. For these, we provide a review that helps readers with general understanding.

The following terms appear repeatedly throughout this article, and thus for convenience, we provide definitions for them here per the taxonomy provided in [37]:

- A failure is when a service deviates from its correct behavior.
- An error is a condition in a system that may lead to a failure.
- A fault is the underlying cause of an error, also known as a bug.

The remainder of this article is organized in the following manner: we begin by describing traditional and intuitive fault localization techniques in Section 2, moving on to more advanced and complex techniques in Section 3. In Section 4, we list some of the popular subject programs that have been used in different case studies and discuss how these programs have evolved through the years. Different evaluation metrics to assess the effectiveness of fault localization techniques are described in Section 5, followed by a discussion of fault localization tools in Section 6. Finally, critical aspects and conclusions are presented in Sections 7 and 8, respectively.

2 Traditional Fault Localization Techniques

This section describes traditional and intuitive fault localization techniques, including program logging, assertions, breakpoints, and profiling.

2.1 Program Logging

Statements (such as *print*) used to produce program logging are commonly inserted into the code in an ad-hoc fashion to monitor variable values and other program state information [105]. When abnormal program behavior is detected, developers examine the program log in terms of saved log files or printed run-time information to diagnose the underlying cause of failure.

2.2 Assertions

Assertions are constraints added to a program that have to be true during the correct operation of a program. Developers specify these assertions in the program code as conditional statements that terminate execution if they evaluate to false. Thus, they can be used to detect erroneous program behavior at runtime. More details of using assertions for program debugging can be found in [309], [310].

2.3 Breakpoints

Breakpoints are used to pause the program when execution reaches a specified point and allow the user to examine the current state. After a breakpoint is triggered, the user can modify the value of variables or continue the execution to observe the progression of a bug. Data breakpoints can be configured to trigger when the value changes for a specified expression, such as a combination of variable values. Conditional breakpoints pause execution only upon the satisfaction of a predicate specified by the user. Early studies (e.g., [80], [155]) use this approach to help developers locate bugs while a program is executed under the control of a symbolic debugger. The same approach is also adopted by more advanced debugging tools such as GNU GDB [121] and Microsoft Visual Studio Debugger [255].

2.4 Profiling

Profiling is the runtime analysis of metrics such as execution speed and memory usage, which is typically aimed at program optimization. However, it can also be leveraged for debugging activities, such as the following:

- Detecting unexpected execution frequencies of different functions (e.g., [43]);
- Identifying memory leaks or code that performs unexpectedly poorly (e.g., [150]);
- Examining the side effects of lazy evaluation (e.g., [313]).

Tools that use profiling for program debugging include GNU's gprof [120] and the Eclipse plugin TPTP [108].

3 Advanced Fault Localization Techniques

With the massive size and scale of software systems today, traditional fault localization techniques are not effective in isolating the root causes of failures. As a result, many advanced fault localization techniques have surfaced recently using the idea of *causality* [215], [288], which is related to philosophical theories with an objective to characterize the relationship between events/causes (program bugs in our case) and a phenomenon/effect (execution failures in our case). There are different causality models [288] such as counterfactual-based, probabilistic- or statistical-based, and causal calculus models. Among these, probabilistic causality models are the most widely used in fault localization to identify suspicious code that is responsible for execution failures.

In this survey, we classify fault localization techniques into eight categories, including slice-based, spectrum-based, statistics-based, program state-based, machine learning-based, data mining-based, model-based and miscellaneous techniques. Many studies that evaluate the effectiveness of specific fault localization techniques have been reported [8], [10], [11], [12], [25], [31], [36], [49], [53], [91], [93], [94], [102], [124], [178], [185], [191], [207], [209], [253], [266], [267], [296], [299], [335], [366], [390], [391], [393], [420], [406], [410], [424]. However, none of them offer a comprehensive discussion on all these techniques.

3.1 Slice-Based Techniques

Program slicing is a technique to abstract a program into a reduced form by deleting irrelevant parts such that the resulting slice will still behave the same as the original program with respect to certain specifications. Hundreds of papers on this topic have been published [52], [344], [394] since Weiser first proposed *static slicing* in 1979 [361].

One of the important applications of static slicing [360] is to reduce the search domain while programmers locate bugs in their programs. This 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, allowing us to confine our search to the slice rather than looking at the entire program. Lyle and Weiser extend 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 [235]. Although static slice-based techniques have been experimentally evaluated and confirmed to be useful in fault localization [207], one problem is that handling pointer variables can make data-flow analysis inefficient because large sets of data facts that are introduced by dereferences of pointer variables need to be stored. Equivalence analysis, which identifies equivalence relationships among the various memory locations accessed by a procedure, is used to improve the efficiency of data-flow analyses in the presence of pointer variables [220]. Two equivalent memory locations share identical sets of data facts in a procedure. As a result, data-flow analysis only needs to compute information for a representative memory location, and data-flow for other equivalent locations can be garnered from the representative location. Static slicing is also applied for fault localization in binary executables [192], and type-checkers [343].

A disadvantage of static slicing is that the slice 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. As a result, it might generate a dice with certain statements that 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* [20], [202] instead of static slicing, as the former can identify the statements that do affect a particular value observed at a particular location, rather than possibly affecting such a value as with the latter. Studies such as [18], [24], [28], [96], [104], [188], [192], [201], [219], [227], [237], [257], [277], [297], [334], [356], [378], [379], [406], [407], [410], which use the dynamic slicing concept in program debugging, have been reported. In [379], Wotawa combines dynamic slicing with model-based diagnosis to achieve more effective fault localization. Using a given test suite against a program, dynamic slices for erroneous variables discovered are collected. Hitting-sets are constructed, which contain at least one statement from each dynamic slice. The probability that a statement is faulty is calculated based on the number of hitting-sets that cover that statement. Zhang et al. [407] propose the multiple-points dynamic slicing technique, which intersects slices of three techniques: Backward Dynamic Slice (*BwS*), Forward Dynamic Slice (*FwS*), and Bidirectional Dynamic Slice (*BiS*). The *BwS* captures any executed statements that affect the output value of a faulty variable, while the *FwS* is computed based on the minimal input difference between a failed and a successful test case, isolating the parts of the input that trigger a failure. The *BiS* flips the values of certain predicates in the execution of a failed test case so that the program generates a correct output. Qian and Xu [297] propose a scenario-oriented program slicing technique. A user-specified scenario is identified as the extra slicing parameter, and all program parts related to a special computation are located under the given execution scenario. There are three key steps to implementing the scenario-oriented slicing technique: scenario input, identification of scenario relevant codes, and, finally, gathering of scenario-oriented slices.

One limitation of dynamic slicing-based techniques is that they cannot capture execution omission errors, which may cause the execution of certain critical statements in a program to be omitted and thus result in failures [411]. Gyimothy et al. [142] propose the use of *relevant slicing* to locate faulty statements responsible for execution omission errors. Given a failed execution, the relevant slicing first constructs a dynamic dependence graph in the same way that classic dynamic slicing does. It then augments the dynamic dependence graph with *potential dependence edges*, and a relevant slice is computed by taking the *transitive closure* of the incorrect output on the augmented dynamic dependence graph. However, incorrect dependencies between program statements may be included to produce oversized relevant slices. To address this problem, Zhang et al. [411] introduce the concept of *implicit dependencies*, in which dependencies can be obtained by predicate switching. A similar idea has been used by Weeratunge et al. [358] to identify root causes of omission errors in concurrent programs, in which *dual slicing*, a combination of dynamic slicing and trace differencing, is used.

An alternative approach to static and dynamic slicing is the use of *execution slicing* based on data-flow tests to locate program bugs [21] in which 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 slicing is that a static slice focuses on finding statements that could possibly have an impact on the variables of interest for *any* inputs, versus statements that are executed by a *specific* input. This implies that a static slice does not make any use of the input values that reveal the fault and violates a very important concept in debugging that suggests programmers analyze the program behavior under the test case that fails and not under a generic test case. Collecting dynamic slices may consume excessive time and file space, even though different algorithms [51], [204], [408], [409] have been proposed to address these issues. Conversely, it is relatively easy to construct the execution slice for a given test case if we collect code coverage data from the execution of the test. Different execution slice-based debugging tools have been developed and used in practice such as χSuds at Telcordia (formerly Bellcore) [22], [427] and eXVantage at Avaya [372]. Agrawal et al. [21] apply the execution slice to fault localization by examining the execution dice of one failed and one successful test to locate program bugs. Jones et al. [186], [187] and Wong et al. [375] extend that study by using multiple successful and failed tests based on the following observations:

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

We use the following example to demonstrate the differences among static, dynamic, and execution slicing. Use the

TABLE 2
An Example Showing the Differences among Static, Dynamic, and Execution Slicing

	Code with a bug at s_7	Static slice for *product*	Dynamic slice for *product* with respect to a test case $a = 2$	Execution slice for *product* with respect to a test case $a = 2$
s_1	input(a)	input(a)	input(a)	input(a)
s_2	$i = 1$;	$i = 1$;	$i = 1$;	$i = 1$;
s_3	$sum = 0$;			$sum = 0$;
s_4	$product = 1$;	$product = 1$;		$product = 1$;
s_5	if ($i < a$){	if ($i < a$){	if ($i < a$){	if ($i < a$){
s_6	$sum = sum + i$;			$sum = sum + i$;
s_7	$product = product \times i$; //bug $product = product \times 2i$	$product = product \times i$;	$product = product \times i$;	$product = product \times i$;
s_8	}else{	}else{		
s_9	$sum = sum - i$;			
s_{10}	$product = product\ /\ i$;	$product = product\ /\ i$;		
s_{11}	}			
s_{12}	print (sum);			print (sum);
s_{13}	print ($product$);	print ($product$);	print ($product$);	print ($product$);

code in column 2 of Table 2 as the reference. Assume it has one bug at s_7. The static slice for the output variable, *product*, contains all statements that could possibly affect the value of *product*, s_1, s_2, s_4, s_5, s_7, s_8, s_{10}, and s_{13}, as shown in the third column. The dynamic slicing for *product* only contains the statements that do affect the value of *product* with respect to a given test case, which includes s_1, s_2, s_5, s_7, and s_{13} (as shown in the fourth column) when $a = 2$. The execution slice with respect to a given test case contains all statements executed by this test. Therefore, the execution slice for a test case, $a = 2$, consists of s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_{12}, $\boldsymbol{s}_{13}$ as shown in the fifth column of Table 2.

One problem with the aforementioned slice-based techniques is that the bug may not be in the dice. Even if a bug is in the dice, there may still be too much code that needs to be examined. To overcome this problem, an inter-block data dependency-based augmentation and a refining method is proposed in [373]. 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. Additionally, slices are problematic because they are always lengthy and hard to understand. In [205], the notion of using *barriers* is proposed to provide a filtering approach for smaller program slices and better comprehensibility. Authors of [330] propose *thin slicing* in order to find only *producer statements* that help compute and copy a value to a particular variable. Statements that explain why producer statements affect the value of a particular variable are excluded from a thin slice.

3.2 Program Spectrum-Based Techniques

Following the discussion in the beginning of Section 3, we would like to emphasize that many spectrum-based techniques are inspired by the probabilistic- and statistical-based causality models. With this understanding, we now explain the details of these techniques.

A program spectrum details the execution information of a program from certain perspectives, such as execution information for conditional branches or loop-free intra-procedural paths [149]. It can be used to track program behavior [305]. An early study by Collofello and Cousins [79] suggests that such spectra can be used for software fault localization. When the execution fails, such information can be used to identify suspicious code that is responsible for the failure. Code coverage, or Executable Statement Hit Spectrum (ESHS), indicates which parts of the program under testing have been covered during an execution. With this information, it is possible to identify which components were involved in a failure, narrowing the search for the faulty component that made the execution fail.

3.2.1 Notation

P	a program
N_{CF}	number of failed test cases that cover a statement
N_{UF}	number of failed test cases that do not cover a statement
N_{CS}	number of successful test cases that cover a statement
N_{US}	number of successful test cases that do not cover a statement
N_C	total number of test cases that cover a statement
N_U	total number of test cases that do not cover a statement
N_S	total number of successful test cases
N_F	total number of failed test cases
t_i	the *i*th test case

3.2.2 Techniques

Early studies [19], [201], [203], [341] only use failed test cases for spectrum-based fault localization, though this approach has subsequently been deemed ineffective [21], [185], [366]. Later studies achieve better results using both the successful and failed test cases and emphasizing the contrast between them. Set union and set intersection are proposed in [303]. The set union 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 excludes the code that is executed by all the successful tests but not by the failed test. Renieris and Reiss [303] propose

TABLE 3
An Example Showing the Suspiciousness Value Computed Using the Tarantula Technique

	Code with a bug at s_7	$a = 0$	$a = 1$	$a = 2$	N_{CF}	N_{CS}	Suspiciousness	Ranking
s_1	input(a)	●	●	●	1	2	0.5	3
s_2	i = 1;	●	●	●	1	2	0.5	3
s_3	*sum = 0;*	●	●	●	1	2	0.5	3
s_4	*product = 1;*	●	●	●	1	2	0.5	3
s_5	if (i < a){	●	●	●	1	2	0.5	3
s_6	*sum = sum + i;*			●	1	0	1	1
s_7	*product = product × i;* *//bug product = product × 2i*			●	1	0	1	1
s_8	}else{	●	●		0	2	0	10
s_9	*sum = sum - i;*	●	●		0	2	0	10
s_{10}	*product = product / i;*	●	●		0	2	0	10
s_{11}	}	●	●		0	2	0	10
s_{12}	print *(sum)*;	●	●	●	1	2	0.5	3
s_{13}	print *(product)*;	●	●	●	1	2	0.5	3
Execution Results		Successful	Successful	Failed				

another ESHS-based technique, nearest neighbor, which contrasts a failed test with a successful test that is most similar to the failed one in terms of the *distance* between them. If a bug is in the difference set, it is located. For a bug that is not contained in the difference set, the process continues by first constructing a program dependence graph (PDG) and then including and checking adjacent un-checked nodes in the graph step by step until all the nodes in the graph are examined. The idea of nearest neighbor is similar to Lewis' counterfactual reasoning [216], which claims that, for two events *A* and *B*, *A* causes *B* (in world *w*) if and only if, in all *possible worlds* that are *maximally similar* to *w*, *A* does not take place and *B* also does not happen. The theory of counterfactual reasoning is also found in other studies such as [137], [179], [400].

Intuitively, the *closer* the execution pattern of a statement is to the failure pattern of all test cases, the more likely the statement is to be faulty, and consequently the more suspicious the statement seems. By the same token, the *farther* the execution pattern of a statement is to the failure pattern, the less suspicious the statement appears to be. Similarity coefficient-based measures can be used to quantify this *closeness*, and the degree of closeness can be interpreted as the suspiciousness of the statements.

A popular ESHS-based similarity coefficient-based technique is Tarantula [186], which uses the coverage and execution results (success or failure) to compute the suspiciousness of each statement as $(N_{CF}/N_F)/(N_{CF}/N_F + N_{CS}/N_S)$. A study on the *Siemens* suite [185] shows that Tarantula inspects less code before the first faulty statement is identified, making it a more effective fault localization technique when compared to others such as set union, set intersection, nearest neighbor and cause transition [77]. Based on the suspiciousness computed by Tarantula, studies like [186], [187] use different colors (from red to yellow to green) to provide a visual mapping of the participation of each program statement in the execution of a test suite. The more failed test cases that execute a statement, the brighter (redder) the color assigned to the statement will be. In [94], Debroy et al. further revise the Tarantula technique. Statements executed by the same number of failed test cases are grouped together, and then groups are ranked in descending order by the number of failed test cases. Using Tarantula, statements are ranked by suspiciousness within each group.

For discussion purposes, let's use the code in Table 2 again. Assume that we have two successful test cases ($a = 0$ and $a = 1$) and one failed test case ($a = 2$). The suspiciousness value of each statement can be computed, for example, using the Tarantula technique discussed above. The results are as shown in Table 3.

The third to fifth columns in Table 3 represent the statement coverage of the three test cases. An entry with a "●" means the statement is covered by the corresponding test case, while an empty entry means the statement is not. The values of N_{CF} and N_{CS} for each statement are given in the sixth and seventh columns. Based on the definition of Tarantula, the suspiciousness value of each statement is computed and displayed in the eighth column. The ranking of each statement is given in the rightmost column. As we can observe, the faulty statement s_7 has the highest ranking.

In recent years, other techniques have also been proposed that perform at the same level with, or even surpass, Tarantula in terms of their effectiveness at fault localization. The Ochiai similarity coefficient-based technique [11] is generally considered more effective than Tarantula, and its formula is as follows:

$$Suspiciousness(Ochiai) = \frac{N_{CF}}{\sqrt{N_F \times (N_{CF} + N_{CS})}}.$$

There are two major differences between Ochiai and the nearest neighbor model: 1) The nearest neighbor model utilizes a single failed test case, while Ochiai uses multiple failed test cases, and 2) The nearest neighbor model only selects the successful test case that most closely resembles the failed test case, while Ochiai includes all successful test cases. Ochiai2 [267] is an extension of Ochiai, and its formula is as follows:

$$Suspiciousness(Ochiai2) = \frac{N_{CF} \times N_{US}}{\sqrt{(N_{CF} + N_{CS}) \times (N_{US} + N_{UF}) \times (N_{CF} + N_{UF}) \times (N_{CP} + N_{US})}}.$$

In [267], Naish et al. propose two techniques, O and O^P (defined as follows). The technique O is designed for programs with a single bug, while O^P is better applied to

TABLE 4
Similarity Coefficient-Based Techniques

	Coefficient	Algebraic Form		Coefficient	Algebraic Form
1	Braun-Banquet	$\frac{N_{CF}}{max(N_{CF}+N_{CS},N_{CF}+N_{UF})}$	17	Harmonic Mean	$\frac{(N_{CF}\times N_{US}-N_{UF}\times N_{CS})((N_{CF}+N_{CS})\times(N_{US}+N_{UF})+(N_{CF}+N_{UF})\times(N_{CS}+N_{US}))}{(N_{CF}+N_{CS})\times(N_{US}+N_{UF})\times(N_{CF}+N_{UF})\times(N_{CS}+N_{US})}$
2	Dennis	$\frac{(N_{CF}\times N_{US})-(N_{CS}\times N_{UF})}{\sqrt{n\times(N_{CF}+N_{CS})\times(N_{CF}+N_{UF})}}$	18	Rogot2	$\frac{1}{4}(\frac{N_{CF}}{N_{CF}+N_{CS}}+\frac{N_{CF}}{N_{CF}+N_{UF}}+\frac{N_{US}}{N_{US}+N_{CS}}+\frac{N_{US}}{N_{US}+N_{UF}})$
3	Mountford	$\frac{N_{CF}}{0.5\times((N_{CF}\times N_{CS})+(N_{CF}\times N_{UF}))+(N_{CS}\times N_{UF})}$	19	Simple Matching	$\frac{N_{CF}+N_{US}}{N_{CF}+N_{CS}+N_{US}+N_{UF}}$
4	Fossum	$\frac{n\times(N_{CF}-0.5)^2}{(N_{CF}+N_{CS})\times(N_{CF}+N_{UF})}$	20	Rogers & Tanimoto	$\frac{N_{CF}+N_{US}}{N_{CF}+N_{US}+2(N_{UF}+N_{CS})}$
5	Pearson	$\frac{n\times((N_{CF}\times N_{US})-(N_{CS}\times N_{UF}))^2}{N_C\times N_U\times N_S\times N_F}$	21	Hamming	$N_{CF}+N_{US}$
6	Gower	$\frac{N_{CF}+N_{US}}{\sqrt{N_F\times N_C\times N_U\times N_S}}$	22	Hamann	$\frac{N_{CF}+N_{US}-N_{UF}-N_{CS}}{N_{CF}+N_{UF}+N_{CS}+N_{US}}$
7	Michael	$\frac{4\times((N_{CF}\times N_{US})-(N_{CS}\times N_{UF}))}{(N_{CF}+N_{US})^2+(N_{CS}+N_{UF})^2}$	23	Sokal	$\frac{2(N_{CF}+N_{US})}{2(N_{CF}+N_{US})+N_{UF}+N_{CS}}$
8	Pierce	$\frac{(N_{CF}\times N_{UF})+(N_{UF}\times N_{CS})}{(N_{CF}\times N_{UF})+(2\times(N_{UF}\times N_{US}))+(N_{CS}\times N_{US})}$	24	Scott	$\frac{4(N_{CF}\times N_{US}-N_{UF}\times N_{CS})-(N_{UF}-N_{CS})^2}{(2N_{CF}+N_{UF}+N_{CS})(2N_{US}+N_{UF}+N_{CS})}$
9	Baroni-Urbani & Buser	$\frac{\sqrt{(N_{CF}\times N_{US})}+N_{CF}}{\sqrt{(N_{CF}\times N_{US})}+N_{CF}+N_{CS}+N_{UF}}$	25	Rogot1	$\frac{1}{2}(\frac{N_{CF}}{2N_{CF}+N_{UF}+N_{CS}}+\frac{N_{US}}{2N_{US}+N_{UF}+N_{CS}})$
10	Tarwid	$\frac{(n\times N_{CF})-(N_F\times N_C)}{(n\times N_{CF})+(N_F\times N_C)}$	26	Kulczynski	$\frac{N_{CF}}{N_{UF}+N_{CS}}$
11	Ample	$\left\lvert\frac{N_{CF}}{N_{CF}+N_{UF}}-\frac{N_{CS}}{N_{CS}+N_{US}}\right\rvert$	27	Anderberg	$\frac{N_{CF}}{N_{CF}+2(N_{UF}+N_{CS})}$
12	Phi (Geometric Mean)	$\frac{N_{CF}\times N_{US}-N_{UF}\times N_{CS}}{\sqrt{(N_{CF}+N_{CS})\times(N_{CF}+N_{UF})\times(N_{CS}+N_{US})\times(N_{UF}+N_{US})}}$	28	Dice	$\frac{2N_{CF}}{N_{CF}+N_{UF}+N_{CS}}$
13	Arithmetic Mean	$\frac{2(N_{CF}\times N_{US}-N_{UF}\times N_{CS})}{(N_{CF}+N_{CS})\times(N_{US}+N_{UF})+(N_{CF}+N_{UF})\times(N_{CS}+N_{US})}$	29	Goodman	$\frac{2N_{CF}-N_{UF}-N_{CS}}{2N_{CF}+N_{UF}+N_{CS}}$
14	Cohen	$\frac{2(N_{CF}\times N_{US}-N_{UF}\times N_{CS})}{(N_{CF}+N_{CS})\times(N_{US}+N_{CS})+(N_{CF}+N_{UF})\times(N_{UF}+N_{US})}$	30	Jaccard	$\frac{N_{CF}}{N_{CF}+N_{UF}+N_{CS}}$
15	Fleiss	$\frac{4(N_{CF}\times N_{US}-N_{UF}\times N_{CS})-(N_{UF}-N_{CS})^2}{(2N_{CF}+N_{UF}+N_{CS})+(2N_{US}+N_{UF}+N_{CS})}$	31	Sorensen-Dice	$\frac{2N_{CF}}{2N_{CF}+N_{UF}+N_{CS}}$
16	Zoltar	$\frac{N_{CF}}{N_{CF}+N_{UF}+N_{CS}+\frac{10000\times N_{UF}\times N_{CS}}{N_{CF}}}$			

programs with multiple bugs. Data from their experiments suggest that O and O^P are more effective than Tarantula, Ochiai, and Ochiai2 for single-bug programs. On the other hand, Le et al. [210] present a different view by showing that Ochiai can be more effective than O and O^P for programs with single bugs

$$Suspiciousness(O) = \langle \begin{matrix} -1, & \text{if } N_{UF} > 0 \\ N_{US}, & \text{otherwise.} \end{matrix}$$

Table 4 lists 31 similarity coefficient-based techniques, along with their algebraic forms, which have been used in different studies such as [75], [364], [371]. A few additional techniques using similar approaches can be found in [230]. Tools like Zoltar [170] and DEPUTO [9] are available to compute the suspiciousness with respect to selected techniques.

Empirical studies have also shown that techniques proposed in [366], [367], [369], [370], [371] are, in general, more effective than Tarantula. Especially in the case of DStar [371], results from empirical evaluations against all 31 similarity coefficient-based techniques listed in Table 4 – as well as Tarantula, Ochiai, Ochiai2, Crosstab [369], H3b and H3c [366], and RBF [367] – suggest that DStar outperforms all compared techniques in most cases.

Comparisons among different spectrum-based fault localization techniques are frequently discussed in recent studies [12], [210], [267], [371]. However, there is no technique claiming that it can outperform all others under every scenario. In other words, an optimum spectrum-based technique does not exist, which is supported by Yoo et al.'s study [397].

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

- Program Invariants Hit Spectrum (PIHS)-based: This spectrum records the coverage of program invariants [107], which are the program properties that remain unchanged across program executions. PIHS-based techniques try to find violations of program properties in failed program executions to locate bugs. *Potential invariants* [294], also called *likely invariants* [317], are program properties that are observed to hold in some sets of successful executions but, unlike *invariants*, may not necessarily hold for all possible executions. The major obstacle in applying such techniques is how to automatically identify the necessary program properties required for the fault localization. To address this problem, existing PIHS-based techniques often take the invariant spectrum of successful executions as the program properties. In study [27], Alipour and Groce propose *extended invariants* by adding execution features such as the execution count of blocks to the invariants. They claim that extended invariants are helpful in fault localization.
- Predicate Count Spectrum (PRCS)-based: PRCS records how predicates are executed and can be used to track program behaviors that are likely to be erroneous. These techniques are often labeled as *statistical debugging* techniques because the PRCS information is analyzed using statistical methods. Fault

TABLE 5
Additional Program Spectra Relevant to 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

localization techniques in this category include Liblit05 [222], SOBER [223] etc. See Section 3.3 for more details. Authors of [266] suggest that using PRCS could achieve a better fault localization effectiveness than that using ESHS.

- Method Calls Sequence Hit Spectrum (MCSHS)-based: Information regarding the sequences of method calls covered during program execution is collected. In one study, Dallmeier et al. [84] 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. In another study, Liu et al. [225] construct software behavior graphs from collected program execution data, including the calling and transition relationships between functions. They define a framework to mine closed frequent graphs based on behavior graphs and use them to train classifiers that help identify suspicious functions.
- Time Spectrum-based: A time spectrum [396] records the execution time of every method in successful or failed executions. Observed behavior models are created using time spectra collected from successful executions. Deviations from these models in failed executions are identified and ranked as potential causes of failures.

Other program spectra such as those in Table 5 [149] can also be applied to identify suspicious code in a program.

3.2.3 Issues and Concerns

A variety of issues and concerns about spectrum-based fault localization has also been identified and studied in depth. One problem is that most spectrum-based techniques do not calibrate the contribution of failed and successful tests. In [385], all statements are divided into suspicious and unsuspicious groups. The suspicious group contains statements that have been executed by at least one failed test case, while the unsuspicious group contains the remaining statements. Risk is only calculated for suspicious statements, and unsuspicious statements are simply assigned the lowest value. It is possible, however, that successful test cases may also contain bugs. In [366], Wong et al. focus on the question of how each additional failed or successful test case can aid in locating program bugs. They describe that with respect to a piece of code, the contribution of the first failed test case that executes it in computing its suspiciousness 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. In addition, the total contribution from all the successful test cases that execute a statement should be less than the total contribution from all the failed tests that execute it. Recognizing that fault localization often proceeds by comparing information associated with a failed test case to that with a successful test case, Wong and Qi [373] and Guo et al. [140] attempt to answer the question of which successful test case should be selected for comparison, in the interests of more effective fault localization. Choosing the successful test case whose execution sequence is most similar to that of a failed test case, according to a control flow-based difference metric, can minimize the search domain of the fault.

For most spectrum-based techniques, if statements exhibit the same execution pattern, there is a high likelihood that the suspiciousness score assigned to these statements will be exactly the same. Statements with the same suspiciousness will result in ties in the ranking. To break these ties, the information related to statement execution frequency in addition to statement coverage can also be utilized [7], [213]. In [393], Xu et al. evaluate different tie-breaking strategies, including statement order-based strategy, confidence-based strategy, and data dependency-based strategy. Tie-breaking methods will be further discussed in Section 7.6. Another problem is that almost all spectrum-based techniques have assumed that a test oracle exists, which restricts their practical applicability. Thus, Xie et al. [387] propose a fault localization technique based on the integration of metamorphic relations and slices, in which a program execution slice is replaced by a metamorphic slice; an individual test case is replaced by a metamorphic test group; and the success/failure result of a test case is replaced by the violation/non-violation result of a metamorphic test group. Authors of [71] also use metamorphic relations with symbolic testing for program debugging. However, all these techniques rely strongly on the metamorphic relations derived from program specifications. Proper identification of such relations can be not only difficult but also time consuming in practice.

Zhao et al. [421], [422] posit that using only individual coverage information may not reveal the execution paths. Therefore, they first use the program control-flow graph to analyze the program execution and then map the distribution of failed executions to different control flows. They use *bug proneness* to qualify how each block contributes to the failure and *bug free confidence* to quantify the likelihood of each block being bug-free by comparing the distributions of blocks on the same failed execution path.

Instrumentation overhead is another issue, which introduces a considerable cost in the fault localization process, especially in a resource-constrained environment. In order to mitigate this problem, Perez et al. [291] propose coined dynamic code coverage by using coarser instrumentation to reduce such overhead. This technique starts by analyzing coverage traces for large components of the program (e.g., package or class) and then progressively increases the instrumentation granularity for possible faulty components until the statement level is reached.

3.3 Statistics-Based Techniques

A statistical debugging technique (Liblit05) that can isolate bugs in programs with instrumented predicates at particular points is presented in [222]. For each predicate P, Liblit05 first computes the probability that P being true implies failure, *Failure*(P), and the probability that the execution of P implies failure, *Context*(P). Predicates that have *Failure*(P) – *Context*(P) ≤ 0 are discarded. The remaining predicates are prioritized based on their *importance* scores, which give an indication of the relationship between predicates and program bugs. Predicates with a higher score should be examined first. Chilimbi et al. [74] propose that replacing predicates with path profiles may improve the effectiveness of Liblit05. Path profiles are collected during execution and are aggregated across the execution of multiple test cases through feedback reports. The *importance score* is calculated for each path and the top results are selected and presented as potential root causes.

In [223], Liu et al. propose the SOBER technique to rank suspicious predicates. A predicate P can be evaluated as true more than once in the execution of one test case. Compute $\pi(P) = \frac{n(t)}{n(t)+n(f)}$, the probability that P is evaluated as true in each execution of a test case, where $n(t)$ is the number of times P is evaluated as true and $n(f)$ is the number of times P is evaluated as false. If the distribution of $\pi(P)$ in failed executions is significantly different from that in successful executions, then P is related to a fault. Hu et al. [168] use a similar heuristic to rank all predicates. In addition, they apply non-parametric hypothesis testing to determine the degree of difference between the spectra of predicates for successful and failed test cases. This new enhancement has been empirically evaluated to be effective [416], [420].

The study in [369] presents a cross tabulation (a.k.a. Crosstab) analysis-based technique to compute the suspiciousness of statements. A crosstab is constructed for each statement with two vertical categories (covered/not covered) and two horizontal categories (successful execution/failed execution). A hypothesis test is used to provide a reference of dependency/independency between the execution results and the coverage of each statement. The exact suspiciousness of each statement depends on the degree of association between its coverage and the execution results.

The primary difference between Crosstab, SOBER, and Liblit05 is that Crosstab can be generally applied to rank suspicious program elements (i.e., statement, predicate, function/method, etc.), whereas the last two only rank suspicious predicates for fault localization. For Liblit05 and SOBER, 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 [185], Liblit05 provides no way to quantify the ranking for all statements. An ordering of the predicates is defined, but the approach does not detail 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 potentially be time consuming. However, such a search is not required for Crosstab, as all the statements of the program are ranked based on their suspiciousness. Results reported in [369] suggest that Crosstab is almost always more effective in locating bugs in the Siemens suite than Liblit05 and SOBER.

In program execution, *short-circuit evaluation* may occur frequently, which means, for a predicate with more than one condition, if the first condition suffices to determine the results of the predicate, the following conditions will not be evaluated (executed). Zhang et al. [414], [415] identify the short-circuit evaluations of an individual predicate and produce one set of evaluation sequences for each predicate. Using such information, their proposed *Debugging through Evaluation Sequences* (DES) approach is compared to existing predicated-based techniques such as SOBER and Liblit05. You et al. [398] propose a statistical approach employing the behavior of two sequentially connected predicates in the execution. They construct a weighted execution graph for each execution of a test case with predicates as vertices and the transition of two sequential predicates as edges. For each edge, a suspiciousness value is calculated to quantify its fault-relevant likelihood. Authors of [38] apply *causal-inference* techniques to the problem of fault localization. A linear model is built on program control-flow graphs to estimate the causal effect of covering a given statement on the occurrence of failures. This model is able to reduce *confounding bias* and thereby help generate better fault localization rankings. In [39], they further enhance the linear model toward better fault localization effectiveness by including information on data-flow dependence. In [256], Modi et al. explore the usage of *execution phase* information such as cache miss rates, CPU and memory usages in statistical program debugging. They suggest coupling *execution phases* with predicates results in higher bug localization accuracy as opposed to when phase information is not used.

3.4 Program State-Based Techniques

A program state consists of variables and their values at a particular point during program execution, which can be a good indicator for locating program bugs. One way to use program states in software fault localization is by relative debugging [4], in which faults in the development version can be located via a runtime comparison of the internal states to a “reference” version of the program. Another approach is to modify the values of some variables to determine which one causes erroneous program execution. Zeller and Hildebrandt propose a technique, delta debugging [400], [401], by contrasting program states between executions of a successful test and a failed test via their memory graphs which are described in [426]. 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. Note that the idea of simplifying failure-inducing inputs discussed in [400], [401] is orthogonal to other techniques, as it significantly reduces the original execution length. The delta tool [86] has been widely used in industry for automated debugging. In [77], Cleve and Zeller extend delta debugging to the cause transition technique to identify the locations and times where the cause of a failure changes from one variable to another. An algorithm named *cts* is proposed to quickly locate cause transitions in a program execution. Similar studies [273], [274] based on combinatorial testing are reported, which separate input parameters into *faulty-possible* and *healthy-possible* and identify minimal failure-inducing combinations of parameters.

However, the cause transition technique is a relatively high-cost approach; there may exist thousands of states in a program execution, and delta debugging at each matching point requires additional test executions to narrow down the causes. Another problem is that the identified locations may not be where the bugs reside. Gupta et al. [141] introduce the concept of a failure-inducing chop as an extension to the cause transition technique to overcome this issue. First, delta debugging is used to identify input and output variables that are causes of failure. Dynamic slices are then constructed for these variables. The code at the intersection of the forward slicing of the input variables and the backward slicing of the output variables is considered suspicious.

Sumner et al. further improve the robustness, precision, and efficiency of delta debugging by combining it with more precise *execution alignment* techniques [338], [339], [389]. However, there are still three limitations to delta debugging: it fails to handle confounding of partial state replacement, it cannot locate execution omission errors, and it suffers from poor efficiency. To address these limitations, Sumner and Zhang [340] propose a cause inference model, *comparative causality*, to provide a systematic technique explaining the difference between a failed execution and a successful execution.

Predicate switching [405], proposed by Zhang et al., is another program state-based fault localization technique where program states are changed to forcefully alter the executed branches in a failed execution. A predicate which, if switched, can make the program execute successfully is labeled as a critical predicate. The technique 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 help determine the next candidates for critical predicates. Wang and Roychoudhury [352] present a similar technique that 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 with outcomes that have been changed are recorded as bugs. A deficiency of predicate switching is that the alternation of program states is never guided by program dependence analysis, even though faults are intrinsically propagated through the chain of program dependences. The study in [217] extends the predicate switching technique and reduces the search space of program states by selecting a subset of trace points in a failed execution based on dependence analysis.

Jeffrey et al. [172] present a value profile-based technique for fault localization to assist developers in software debugging. The approach involves computing Interesting Value Mapping Pairs (IVMPs) that show how values used in particular program statements can be altered so that failed test cases will produce the correct output instead. Alternate sets of values are selected from profiling information taken from the executions of all test cases in an available test suite. Different alternate value sets are used to perform value replacements in each statement instance for every failed test case. Using these IVMPs, each statement can then be ranked according to the number of failed executions in which at least one IVMP is identified for that statement. In [417], Zhang et al. claim that a bug within a statement may propagate a series of *infected program states* before it manifests the failure. Also, even if every failed execution executes a particular statement, this statement is not necessarily the root cause of the failure. Thus, they use edge profiles to represent program executions and assess the suspiciousness of the infected program states propagated through each edge. By associating basic blocks with edges, a suspiciousness ranking is generated to locate program bugs.

3.5 Machine Learning-Based Techniques

Machine learning is the study of computer algorithms that improve through experience. Machine learning techniques are adaptive and robust and can produce models based on data, with limited human interaction. This has led to their employment in many disciplines such as bioinformatics, natural language processing, cryptography, computer vision, etc. In the context of fault localization, the problem at hand can be identified as trying to learn or deduce the location of a fault based on input data such as statement coverage and the execution result (success or failure) of each test case.

Wong and Qi [374] propose a fault localization technique based on a back-propagation (BP) neural network, one of the most popular neural network models in practice [112]. A BP neural network has a simple structure, which makes it easy to implement using computer programs. Also, BP neural networks have the ability to approximate complicated nonlinear functions [154]. The coverage data of each test case and the corresponding execution result are collected, and they are used together to train a BP neural network so that the network can learn the relationship between them. Then, the coverage of a suite of virtual test cases that each covers only one statement in the program is input to the trained BP network, and the outputs can be regarded as the likelihood of each statement containing the bug. Ascari et al. [36] extend the BP-based technique [374] to object-oriented programs. As BP neural networks are known to suffer from issues such as paralysis and local minima, Wong et al. [367] propose another approach based on radial basis function (RBF) networks, which are less susceptible to these problems and have a faster learning rate [211], [357]. The RBF network is trained using an approach similar to the BP network. Once the training is completed, the output with respect to the coverage of each virtual test case is considered to be the suspiciousness of the corresponding statement. There are three novelties of this approach: 1) a method for representing test cases, coverage information, and execution results within a modified RBF neural network formalism, 2) an innovative algorithm to simultaneously estimate the number of hidden neurons and their receptive field centers, 3) a weighted bit-comparison based distance (instead of the Euclidean distance) to measure the distance between the coverage of two test cases.

In [57] Briand et al. use the C4.5 decision tree algorithm to construct rules that classify test cases into various partitions such that failed test cases in the same partition most likely fail due to the same causative 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 used to rank the statements using a heuristic similar to Tarantula [185] to form a ranking. These individual rankings are then consolidated to form a final statement ranking which can be examined to locate the faults.

3.6 Data Mining-Based Techniques

Along the lines of machine learning, data mining also seeks to produce a model using pertinent information extracted from data. Data mining can uncover hidden patterns in samples of data that may not be discovered by manual analysis alone, especially due to the sheer volume of information. 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 to identify the pattern of statement execution that leads to a failure. In addition, although the complete execution trace of a program is a valuable resource for fault localization, the huge volume of data makes it unwieldy for usage in practice. Therefore, some studies have creatively applied data mining techniques to execution traces.

Nessa et al. [269] 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 that is higher than a certain threshold. A statistical analysis is conducted to determine the conditional probability that a certain *N*-gram appears in a given failed execution trace – this probability is known as the *confidence* for that *N*-gram. *N*-grams are sorted in descending order of confidence and the corresponding statements in the program are displayed based on their first appearance in the list. Case studies on the Siemens suite as well as the *space* and *grep* programs have shown that this technique is more effective at locating faults than Tarantula.

Cellier et al. [65], [66] discuss a combination of association rules and Formal Concept Analysis to assist in fault localization. The proposed technique tries to identify rules regarding the association between statement coverage and corresponding execution failures. The frequency of each rule is measured. A threshold is decided upon to indicate the minimum number of failed executions that should be covered by a selected rule. A large number of rules so generated are partially ranked using a rule lattice. The ranking is then examined to locate the fault.

In [403], the authors propose a technique taking advantage of the recent progress in multi-relational data mining for fault localization. More specifically, this technique is based on Markov logic, combining first-order logic and Markov random fields with weighted satisfiability testing for efficient inference and a voted perceptron algorithm for criminative learning. When applied to fault localization, Markov logic combines different information sources such as statement coverage, static program structure information, and prior bug knowledge into a solution to improve the effectiveness of fault localization. Their technique is empirically shown to be more effective than Tarantula on some programs of the Siemens suite.

Denmat et al. [99] propose a technique that re-interprets Tarantula as a data-mining problem. In this technique, association rules that indicate the relationship between a single statement and a program failure are mined based on the coverage information and execution results of a test suite. The relevance values of these rules are evaluated based on two metrics, *conf* and *lift*, which are commonly used by classical data mining problems. Such values can be interpreted as the suspiciousness of a statement that may contain bugs.

3.7 Model-Based Techniques

With respect to each model-based technique, a critical concern is the model's expressive capability, which has a significant impact on the effectiveness of that technique.

While using model-based diagnosis [301], it is assumed that a correct model of each program being diagnosed is available. That is, these models can be served as the *oracles* of the corresponding programs. Differences between the behaviors of a model and the actual observed behaviors of the program are used to help find bugs in the program [249], [250]. On the other hand, for model-based software fault localization [6], [40], [97], [117], [194], [242], [243], [246], [248], [378], [380], [381], [382], models are generated directly from the actual programs, which may contain bugs. Differences between the observed program executions and the expected results (provided by programmers or testers) are used to identify model elements that are responsible for such observed misbehaviors. As demonstrated by the Java Diagnosis Experiments (JADE) in [241], [252], model-based software fault localization can be viewed as an application of model-based diagnosis [14].

Dependency-based models are derived from dependencies between statements in a program, by means of either static or dynamic analysis. Mateis et al. [242] present a functional dependency model for Java programs that can handle a subset of features for the Java language, such as classes, methods, conditionals, assignments, 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. This dependency-based model is then extended to handle unstructured control flows in Java programs [244], [245], such as exceptions, recursive method calls, return and jump statements. The notion of a dependence graph has also been extended to model behaviors of a program over a test suite. Baah et al. [40] use a probabilistic program dependence graph to model the internal behaviors of a program, facilitating probabilistic analysis and reasoning about uncertain program behaviors, especially those that are likely associated with faults.

Wotawa et al. [380] use first order logic to construct dependency-based models based on source code analysis of target programs to represent program structures and behaviors. Test cases with expected outputs are also transformed into observations in terms of first order logic. If the execution of a target program on a test case fails, conflicts between the test case and the models (which can be shown as equivalent to either static or dynamic slices [378]) are used to identify suspicious statements responsible for the failure. For each statement, a default assumption is made to suggest whether the statement is correct or incorrect. These assumptions are to be revised during fault localization until the failure can be explained. The limitation is that their study only focuses on loop-free programs. To fix this problem, Mayer and Stumptner [246] propose an abstraction-based model in which abstract interpretation [55], [78] is applied to handle loops, recursive procedures, and heap data structures. Additionally, abstract interpretation is used to improve the effectiveness of slice-based and other model-based fault localization techniques [247].

In addition to dependency-based and abstraction-based models, value-based models [196], [251] that represent data-flow information in programs are also applied to locate components that contain bugs. However, value-based models are more computationally intensive than dependency-based and are only practical for small programs [250].

We now discuss model checking-based fault localization techniques that rely on the use of model checkers to locate bugs [44], [67], [133], [134], [136], [137], [138], [200]. If a model does not satisfy the corresponding program specifications (implying that the model contains at least one bug), a model checker can be used to provide counter-examples showing how the specifications will be violated. A counter-example does not directly specify which parts of a model are associated with a given bug; however, it can be viewed as a failed test case to help identify the *causality* of the bug [135].

Ball et al. [44] propose to use a model checker to explore all program paths except that of the counter-example. Successful execution paths (those that do not cause a failure) are recorded. An algorithm is used to identify the *transitions* that appear in the execution path of the counter-example but not in any successful execution paths. Program components related to these *transitions* are those that are likely to contain the causes of bugs. This technique suffers from two weaknesses. First, as suggested by Groce and Visser [136], generating all successful execution paths can be very expensive. Second, only one counter-example is used to locate bugs, even though the same bug may be triggered by multiple counter-examples. If this occurs, using only one example can introduce possible bias. To overcome these problems, Groce and Visser [136] generate a small number of executions by exploring backwards from the original counter-example using a model checker. Additional executions so generated may or may not cause a failure. They then analyze the differences (in terms of *transitions*, *invariants*, and *transformations*) between failed and successful executions to identify possible locations of bugs.

Inspired by Lewis' counterfactual reasoning [216], Groce et al. [135], [137] represent program executions as sets of assignments to variables. They then define a distance metric to measure the distance between two program executions. Based on this metric, a model checker is used to generate one successful execution which is *closest* to the counter-example. The differences between the successful execution and the counter-example provide the possible explanations and locations of bugs. A tool, *explain* [138], is used to implement their technique. Chaki et al. further extend the technique of Groce et al. by combining it with *predicate abstraction* [67].

Techniques such as [44], [67], [135], [136], [137], [138] require at least one successful execution. Griesmayer et al. [133], [134] argue that a successful execution path can be very different from the path of the counter-example and cannot be easily identified using the above techniques. Instead of searching for successful execution paths with small changes from that of the original counter-example, they make minimal changes to the program so that the counter-example will not fail in the revised program. Assuming there is only one bug in one program component, Griesmayer et al. propose a technique with two steps: 1) revising the program specification in such a way that if any one component in the original program is changed, then the original specification cannot be satisfied, and 2) creating variants of the original program such that each variant has exactly one component replaced by a different component with an *alternative behavior*. For each variant, if a model checker can find a counter-example violating the revised specification, then the replaced component is potentially responsible for the failure. Since more than one component may be responsible for the failure, programmers have to manually inspect these components to identify the one containing the bug. Experiments in [133] use the model Checker CBMC, whereas extended studies using an additional model checker SATABS are reported in [134].

Based on a similar idea described in [133], [134], Könighofer and Bloem [200] use symbolic execution to locate bugs for imperative programs. An important point stated by Griesmayer [134] is that the extensive use of a model checker makes their techniques less efficient (in terms of time) than those in [44], [67], [136], [137], [138]; however, fault localization using model checkers can be used to refine results from less precise techniques.

Last but not least, the idea of modifying a program so that test cases that fail on the original program can be executed successfully on the modified program [133], [134], [200] is also used in other studies for automatic bug fixing [95], [152], [189], [270].

Additional model-based fault localization techniques also exist. They can be applied to functional programs [336], hardware description languages like VHDL [290], [376], and spreadsheets [163], [169]. Studies such as [272], [377] make use of constraint solving, in which programs are automatically compiled into a set of constraints. In [97], DeMillo et al. propose a model for analyzing software failures and faults for debugging purposes. Failure modes and failure types are defined to identify the existence of program failures and to analyze the nature of program failures, respectively. Failure modes are used to answer the question "How do we know the execution of a program fails?" and failure types are used to answer the question "What is the failure?" When 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 slice) and testing information are then used to reduce the search domain for locating the fault by predicting possible faulty statements. A significant drawback 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 behaviors and symptoms when they fail. As a result, we do not have a complete relationship between all possible failure modes and failure types, and we might not be able to identify possible failure types responsible for the failure being analyzed.

3.8 Additional Techniques

In addition to those discussed above, there are other techniques for software fault localization. Many of them focus on specific program languages or testing scenarios. Listed below are a few examples.

Development of software systems, while enhancing functionality, will inevitably lead to the introduction of new bugs, which may not be detected immediately. Tracing the behavior changes to code changes can be highly time-consuming. Bohnet et al. [54] propose a technique to identify recently introduced changes. Dynamic, static, and code change information is combined to reduce the large number of changes that may have impact on faulty executions of the system. In this way, root cause changes can be semi-automatically located.

In spite of using garbage collection, Java programs may still suffer from memory leaks due to unwanted references. Chen and Chen [69] develop an aspect-based tool, FindLeak, utilizing an aspect to gather memory consumption statistics and object references created during a program execution. Collected information is then analyzed to help detect memory leaks.

An implicit social network model is presented in [70] to predict possible locations of faults using fault locations cited by similar historical bug reports retrieved from BRMS (bug report managing systems).

In [88], de Souza and Chaim propose a technique using integration coverage data to locate bugs. By ranking the most suspicious pairs of method invocations, *roadmaps*, which are sorted lists of methods to be investigated, are created.

Gong et al. [123] propose an interactive fault localization technique, TALK, which incorporates programmers' feedback into spectrum-based fault localization techniques. Each time a programmer inspects a suspicious program element in the ranking generated by a fault localization technique, he or she can judge the correctness of the element and provide this information as feedback to re-order the ranking of elements that are not yet inspected. The authors demonstrate that using programmers' feedback can help increase the effectiveness of existing fault localization techniques.

To better understand a program's behavior, software developers must translate their questions into code-related queries, speculating about the causes of faults. Whyline [195] is a debugging tool that avoids such speculation by enabling developers to select from a set of "why did" and "why didn't" questions derived from source code. Using a combination of static and dynamic slicing, and precise call graphs, the tool can find possible explanations of failures.

Authors of [72] propose a software fault localization technique that mines bug signatures within a program. A bug signature is a set of program elements that are executed by most failed tests but not by successful tests in general. Bug signatures are ranked in descending order by a discriminative significance score indicating how likely it is to be related to the bug. This ranking is used to help identify the location of the bug.

Maruyama et al. [238] indicate that the culprit of an overwritten variable is always the last write-access to the memory location where the bug first appeared. Removing such bugs begins with finding the last write, followed by moving the control point of execution back to the time when the last write was executed. Generally, the statement that makes the last write will be faulty.

Recently, some studies [85], [234], [298], [315], [425] have applied information retrieval techniques to software fault localization. These studies use an initial bug report to rank the source code files in descending order based on their relevance to the bug report. The developers can then examine the ranking and identify the files that contain bugs. Unlike spectrum-based fault localization techniques, information retrieval-based techniques do not require program coverage information, but their generated ranking is based solely on source code files rather than on program elements with finer granularity such as statements, blocks, or predicates.

Algorithmic debugging (also called declarative debugging), first discussed in Shapiro's dissertation [325] with more details in [328], [402], decomposes a complex computation into a series of sub-computations to help locate program bugs. The outcome of each sub-computation is checked for its correctness with respect to given input values. Based on this, an algorithmic debugger is used to identify a portion of code that may contain bugs. One issue of applying this technique in practice is that testing oracles may not available for sub-computations.

Formula-based fault localization techniques [76], [109], [180], [181] rely on an encoding of failed execution traces into *error trace formulae*. By proving the unsatisfiability of an error trace formula using certain tools or algorithms, the programmer may capture the relevant statements causing the failure. Jose and Majumdar [180], [181] propose a technique, BugAssist, which uses a MAX-SAT solver to compute the maximal set of statements that may cause the failure from a failed execution trace. In [109], Ermis et al. introduce *error invariants*, which provide a semantic argument as to why certain statements of a failed execution trace are irrelevant to the root cause of the failure. By removing such statements, the bug can be located with less manual effort. A common weakness of these techniques [109], [180], [181] is that they only report a set of statements that may be responsible for the failure without providing the exact input values that make the executions go to those statements.

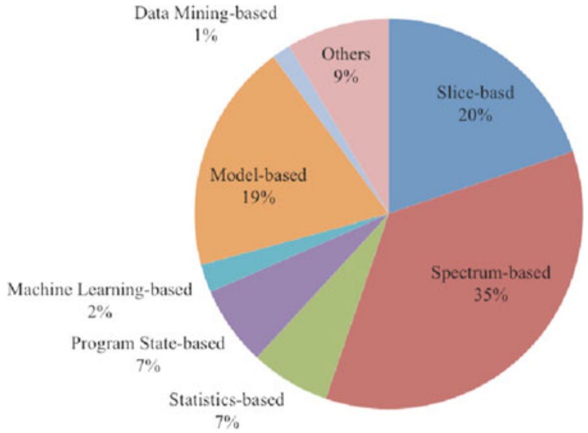


Fig. 3. Distribution of papers in our repository.

Christ et al. [76] address this problem by reporting an extended study based on error invariants [109] that encodes a failed execution trace into a *flow-sensitive error trace formula*. In addition to providing a set of statements that are relevant to the failure, they also specify how these statements can be executed using different input values.

During program maintenance, source code may be modified to fix bugs or enhanced to support new functionalities. Regression testing is also conducted to prevent invalidation of previously tested functionality. If an execution fails, the programmer needs to find the failure-inducing changes. Crisp [302] is a tool to build a compliant intermediate version of the program by adding a partial edit (i.e., a subset of recent changes) to the code before the maintenance is performed. This tool helps programmers focus on a specific portion of changes in the code during the debugging.

Concurrent programs are becoming more prevalent in applications that affect our everyday lives. However, due to their non-determinism, it is very difficult to debug these programs. It is proposed that injecting random timing noise into many points within a program can assist in eliciting bugs. Once the bug is triggered, the objective is to identify a small set of points that indicate the source of the bug. In [398], the authors propose an algorithm that iteratively samples a lower dimensional projection of the program space and identifies candidate relevant points. Refer to Section 7.7 for more discussion.

3.9 Distribution of Papers in Our Repository

Fig. 3 shows the distribution of papers in our repository across all categories. Spectrum-based is the most dominant category with 35 percent of all the papers[4] followed by slice-based, which contains 20 percent, and model-based, which contains 19 percent. The number of papers in each of the statistics-based, program state-based, and others categories is between 7 and 9 percent. The data-mining and machine learning-based categories have the fewest number of papers with only 1 and 2 percent.

Below we present the distribution using a different classification: static and dynamic slice-based, execution slice and program spectrum-based, and other techniques (see Footnote 4 for the rationale). Fig. 4 gives the number of papers published each year with respect to this new classification. The first (leftmost) bar gives the total number of papers from 1977 to 1995, the last (rightmost) only counts papers between January and November 2014, and those in between give the number in the corresponding year. Fig. 5 displays the information from a cumulative point of view. Each data point gives the cumulative number of papers published up to the corresponding year. From these two figures, we make the following observations:

- Static and dynamic slice-based techniques were popular between 2002 and 2007. However, the number of papers each year in this category has decreased since then.
- The number of papers on execution slice and program spectrum-based techniques has increased dramatically since 2008, indicating that more studies are focused on these techniques rather than static or dynamic slice-based techniques in the recent years.

4 Subject Programs

Table 6 presents a list of popular subject programs used to study the effectiveness of different fault localization techniques. This table gives the name, the size (lines of code), a brief description of the functionality, the programming language, and the number of papers that use this program.

We notice that the Siemens suite is the most frequently used. However, every program in the suite is very small-sized with less than 600 lines of code (not including blank lines). Another important point worth noting is that most of the bugs used in the experiments are mutation-based artificially injected bugs. Although mutation has been shown to be an effective approach to simulate realistic faults [29], [103], [223], [268], some real-life bugs are very delicate and cannot be modeled by simple first-order mutants.

With the introduction of advanced techniques in software fault localization, more accurate cross comparisons of their effectiveness are in demand. Furthermore, the feasibility of a technique and the benefits of using it should be demonstrated in an industry-like environment, in contrast to an academic laboratory-oriented controlled environment. In response to these challenges, more and more studies use larger and complex programs in their experiments. Another trend is to use bugs actually introduced at the development phase such as those from Bugzilla for the gcc program and the bugs for Mozilla firefox.

5 Evaluation Metrics

Since a program bug may span multiple lines of code, which are not necessarily contiguous or in the same module, the examination of suspicious code stops as long as one faulty location is identified. This is because the focus is to help programmers find a good starting point to initiate the bug-fixing process rather than to provide the complete set of code that must be modified, deleted, or added with respect to each bug. With this in mind, the effectiveness of a software fault localization technique is defined as the

[4] Papers that only use execution slice-based techniques (e.g., [21], [373]) are included in the spectrum-based category because a statement-based execution slice is the same as ESHS (Refer to Section 3.2). The slice-based category contains papers only using static slicing and/or dynamic slicing.

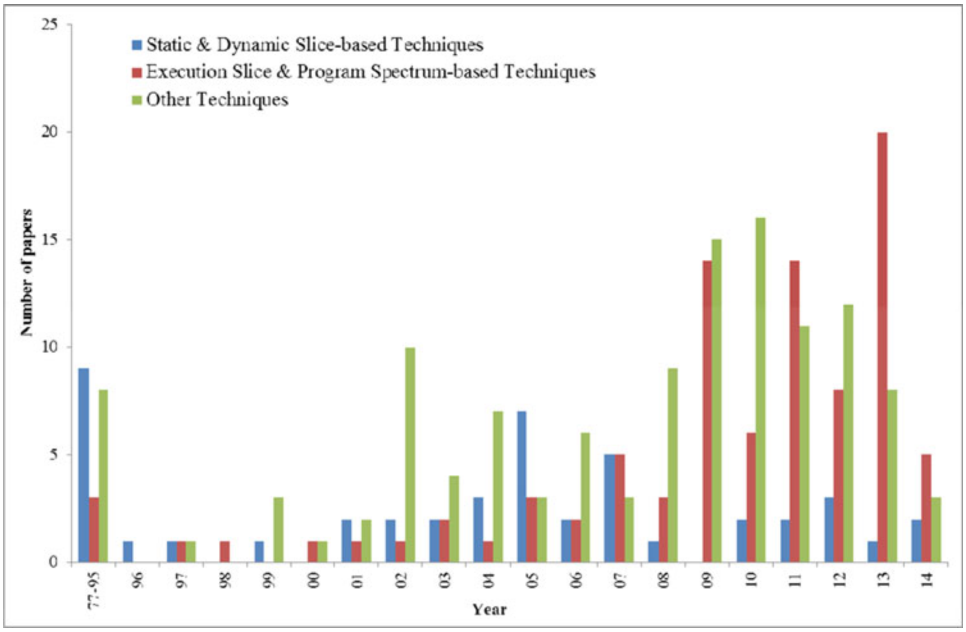


Fig. 4. Number of papers published each year with respect to three different categories.

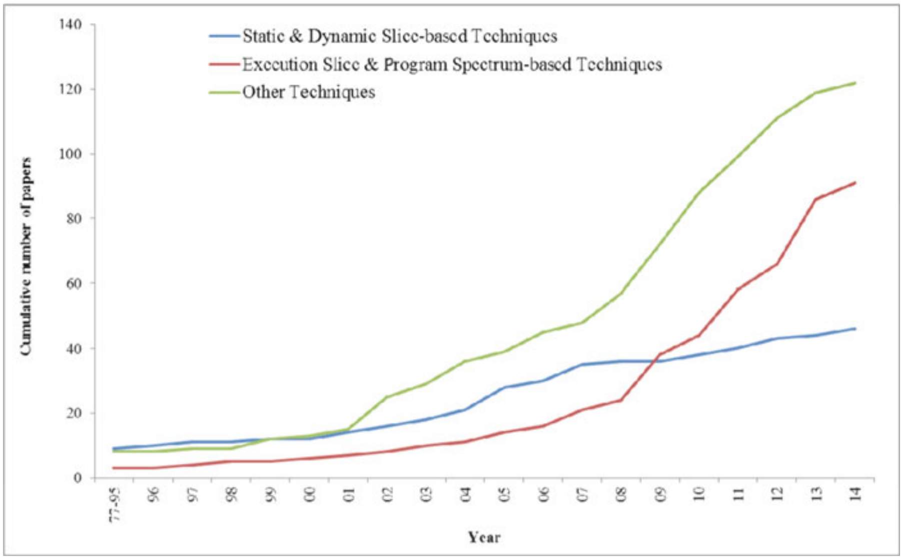


Fig. 5. Cumulative number of papers with respect to three different categories.

percentage of code[5] that needs to be examined before the first faulty location for a given bug is identified.

The T-score [223], [303] estimates the percentage of code a programmer need not examine before the first faulty location is found. A program dependence graph is constructed, and the nodes are marked as *faulty* if they are reported by differencing the correct and the faulty versions of the program, and *blamed* if they are reported by the localizer. For a node n, the corresponding *k-dependency sphere set* (DS_k) is the set of nodes for which there is a directed path of length no more than k that joins n and them. For example, DS_0 contains the node n itself. DS_1 includes not only n but also all the nodes such that there is an edge from them to n, or from n to them. For a report R (i.e., a set of nodes the localizer indicates as possible locations of the bug), let $DS_*(R)$ be the smallest dependency sphere that includes a faulty node. The T-score of a given R is computed using the ratio of the number of nodes in its smallest dependency sphere to the number of nodes in the entire PDG:

$$T\text{-}score = 1 - \frac{|DS_*(R)|}{|PDG|}.$$

The use of T-score requires that programmers are able to distinguish defects from non-defects at each location and can do so at the same cost for each location considered [77]. Furthermore, it assumes that programmers can follow the control- and/or data-dependency relations among statements while searching for faults.

The EXAM [188], [366], [367], [369], [374] or *Expense* [185] score is the percentage of statements in a program that has to be examined until the first faulty statement is reached:

[5] Code can be represented in terms of statements, predicates, functions, etc.

TABLE 6
Summary of Popular Subject Programs Used in the Fault Localization Studies

Name	Size (Lines of code)	Brief Description	Language	Number of papers
Siemens: tcas	173	Altitude separation	C	98
Siemens: schedule	412	Priority scheduler	C	96
Siemens: print_tokens	565	Lexical analyzer	C	96
Siemens: replace	563	Pattern recognition	C	94
Siemens: print_tokens2	510	Lexical analyzer	C	92
Siemens: schedule2	307	Priority scheduler	C	92
Siemens: tot_info	406	Information measure	C	91
grep	12,653	Command-line utility for searching plain-text data sets	C	35
space	9,126	ADL Interpreter	C	34
gzip	6,573	Data compression	C	34
sed	12,062	GNU batch stream editor	C	19
flex	13,892	Lexical analyzer generator	C	17
NanoXML	7,646	XML parser	Java	16
Unix: Cal	202	Print a calendar for a specified year or month	C	13
Unix: Col	308	Filter reverse line	C	13
Unix: Tr	137	Translate characters	C	13
Unix: Spline	338	Interpolate smooth curves based on given data	C	12
Unix: Uniq	143	Report or remove adjacent duplicate lines	C	12
Unix: Chckeq	102	Report missing or unbalanced delimiters and .EQ/.EN pairs	C	11
make	20,014	Manage building of executable and other products from code	C	10
Ant	75,333	Java applications builder	Java	10
XML-sec	21,613	library for XML encryption	C	9
Unix: Look	170	Find words in the system dictionary or lines in a sorted list	C	7
Unix: Comm	167	Select or reject lines common to two sorted files	C	6
tar	25,854	Tool to create file archives	C	6
DC	2,700	reverse-polish desk calculator	Java	5
Unix: Crypt	134	Encrypt and decrypt a file using a user supplied password	C	5
Unix: Sort	913	Sort and merge files	C	5
gcc	222,196	GNU C compiler	C	5
apache	85,661	Http server for hosting web applications	C	5
schoolmate	4,263	A PHP/MySQL solution for administering schools	PHP	4
faqforge	734	A tool for creating and managing documents	PHP	4
webchess	2,226	An online chess game	JS and PHP	4
jtopas	5,400	Text parser	Java	4
timeclock	13,879	A web-based clock system	C	3
phpsysinfo	7,745	Displays system information, e.g., uptime, CPU, memory, etc.	C	3
TCC	1,900	A small and fast compiler for the C programming language	C	3
Xerces	52,528	XML parser	C++	3
Mozilla firefox	3,4M	Web browser	C and C++	3
tidy	31,132	A text editor for editing web content	C++	3

$$\text{EXAM}score = \frac{\text{Number of statements examined}}{\text{Total number of statements in the program}} \times 100\%.$$

In [185], the authors use the executable statements instead of the total number of statements. For techniques such as [224] that generate a ranking of predicates (instead of statements) sorted in descending order of their fault relevance, the EXAM score can also be computed in terms of percentage of predicates that need to be examined. The P-score [420], defined as follows, uses the same approach:

$$P\text{-}score = \frac{1 - based\ index\ of\ P\ in\ L}{number\ of\ predicates\ in\ L} \times 100\%,$$

where L is a list of sorted predicates as described above, P is the most fault-relevant predicate to a fault, and the notation of 1-based index means the first predicate of L is indexed by 1 (rather than 0). Studies in [366], [367], [369], [371], [374] also provide figures that report the percentage of all the faulty versions of a given program in which faults can be located by the examination of an amount of code less than or equal to a given EXAM score. A similar idea is subsequently used by Gong et al. to define the N-score [124]:

$$N\text{-}score = \frac{N_{\det ected}}{N_{statistic}} \times 100\%.$$

When compared to T-score, EXAM is easier to understand, as it is directly proportional to the amount of code to be examined rather than to an indirect measurement in terms of the amount of code that does not need to be examined (as what T-score does). In summary, the lower the EXAM score (or *Expense* or P-score), the more effective the technique, whereas it is the opposite for the T-score (i.e., the lower the T-score, the less effective the technique).

The Wilcoxon signed-rank test (an alternative to the paired Student's t-test when a normal distribution of the population cannot be assumed) can also be used as a metric to present an evaluation from a statistical point of view [370], [371]. If we assume a technique α is more effective than another technique β, we examine the one-tailed alternative hypothesis that β requires the examination of an equal or greater number of statements than α. The confidence with which the alternative hypothesis can be accepted helps us determine whether α is statistically more effective than β. Another metric is the total (cumulative) number of statements that need to be examined to locate all bugs of a given scenario [366], [367], [369], [371]. This metric gives a global view in contrast to the Wilcoxon test, which focuses more on individual pairwise comparisons.

An effective fault localization technique should assign a unique suspiciousness value to each statement; in practice, however, the same suspiciousness may be assigned to different statements. If this happens, two different levels of effectiveness result: 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. The *worst* effectiveness occurs if the faulty statement is the last to be examined. Reporting only the worst case (such as [28], [165]) or only the best case (such as the P-score in [420]) may not give the complete picture because it is very unlikely that programmers will face the worst or the best case scenario in practice. In most cases, they will see something between the best and the worst. It is straightforward to compute the average effectiveness from the best and worst effectiveness. However, the converse is not true. Providing the average effectiveness offers no insights on where the best and worst effectiveness may lie, and, more seriously, can be ambiguous and misleading. For example, two techniques can have the same average effectiveness, but one has a smaller range between the best and the worse cases while the other has a much wider range. As a result, these two techniques should not be viewed as equally effective as suggested by their average effectiveness. Thus, a better approach is to report the effectiveness for both the best and the worst cases such as [366], [367], [369], [374] and perform the cross evaluation under each scenario.

All the evaluation metrics discussed above are based on an assumption of perfect bug detection, which is the same as having an *ideal user* [303] to examine suspicious code to determine whether it contains bugs. That is, a bug in a statement will be detected if the statement is examined. However, a recent study [285] indicates that such an assumption does not always hold in practice. If so, then the number of statements that need to be examined to find the bug may increase.

There are other factors that may affect the effectiveness of a software fault localization technique. Bo et al. [53] present a metric, *Relative Expense*, to study the impact of test set size on the *Expense* score. More discussion regarding the impact of test cases on fault localization appears in Section 7.2. Monperrus [258] suggests that effectiveness should be evaluated with respect to different classes of faults. It is possible that one technique is more effective than another for bugs that can be triggered consistently under some well-defined conditions (namely, Bohrbugs in [139]), but less effective for bugs whose failures cannot be systematically reproduced (namely, Mandelbugs). Instrumentation overhead, interference within multiple bugs, and programming language also have an impact on effectiveness of fault localization [90], [333].

Last but not least, it is important to realize that software fault localization techniques should not be evaluated only in terms of effectiveness as described above [285]. Other factors such as computational overhead, time and space for data collection, amount of human effort, and tool support need also be considered. In addition, we also need to emphasize user-centered aims such as how programmers actually debug, how they reveal the cause-effect chains of failures, and how they decide upon solutions beyond a suspiciousness ranking of code. Unfortunately, none of the published studies has reported a comprehensive evaluation covering all these aspects.

6 Software Fault Localization Tools

One challenge for many empirical studies on software fault localization is that they require appropriate tool support for automatic or semi-automatic data collection and suspiciousness computation. Table 7 gives a list of commonly used tools, including name, a brief description, availability, and which papers use the tool. Of the 63 tools, two are commercial, 16 are open source, 10 are openly accessible but the source code is not available, and the rest may be acquired by contacting their authors.

7 Critical Aspects

In this section, we explore some critical aspects of software fault localization.

7.1 Fault Localization with Multiple Bugs

The majority of published papers in software fault localization focus on programs with a single bug (i.e., each faulty program has exactly one bug). However, this is not the case for real-life software, which in general contains multiple bugs. Results of a study [143] based on an analysis of fault and failure data from two large, real-world projects show that individual failures are often triggered by multiple bugs spread throughout the system. Another study [231] also reports a similar finding. This observation raises doubts concerning the validity of some heuristics and assumptions based on the single-bug scenario. In response, studies have been conducted using programs with multiple bugs [87], [90], [101], [102], [125], [172], [173], [184], [224], [293], [331], [332], [359], [395], [423].

TABLE 7
Summary of Tools Used in the Fault Localization Studies

Name	Brief Description	Availability	Papers using the tool
Ample	Eclipse plug-in for identifying faulty classes in Java program	openly accessible	[12]
Apollo	Automatic tool that efficiently finds and localizes malformed HTML and execution failures in web applications that execute PHP code on the server side	via author	[33]
Atomizer	A dynamic atomicity checker	via author	[116]
ATAC/χslice	Slicing and dicing tool for ANSI C programs	via author	[21], [96], [187]
BARINEL	Framework to combine spectrum-based fault localization and model-based diagnosis	via author	[14]
BugAssist	Fault localization tool for ANSI-C programs	via author	[180], [181]
BugFix	A machine learning-based tool for program debugging	via author	[174]
Chianti	Impact analyzer of program changes for Java programs	via author	[73]
Chislice	Execution slicing tool	via author	[21]
Chord	Debugging tool for concurrent program	via author	[262]
CIL framework	Tool for extracting control flow graph and data flow information from C programs	via author	[40]
Clover	Tool for collecting execution trace information for Java programs	commercial	[91]
CnC	Static checking and testing tool	openly accessible	[81]
CPTEST	A Framework for Automatic Fault Detection, Localization & Correction of Constraint Programs	via author	[208]
Crisp	Eclipse plug-in for constructing intermediate versions of a Java program that is being edited	via author	[73]
Daikon	Dynamic invariant detector	open source	[58], [107]
DejaVu	Regression test selection tool	via author	[149]
Delta	Tool for delta debugging	open source	[86]
Diablo	A link-time optimizer	open source	[141]
DiffJ	Tool for comparing different versions of programs to find bugs	open source	[28]
Doxygen	Source code documentation generator and static analysis	open source	[54]
DrDebug	Debugging tool integrating dynamic slicing and gdb debugger	open source	[104], [356]
ESC/Java	Compile-time program checker to detect precondition violations	open source	[81]
FindLeaks	Aspect-based tool to locate memory leaks in Java programs	via author	[69]
Gcov	Profiling tool to collect program spectra	open source	[13], [53], [102], [213]
GNU GDB	A debugger developed by GNU	open source	[121]
gprof	GNU's profiling tool	open source	[120]
GoalDebug	Constraint-based spreadsheet debugging tool	via author	[2]
GZoltar	An automated testing and debugging framework	openly accessible	[236], [130]
HOLMES	Statistical debugging tool	via author	[74]
HSFal	hybrid slice spectrum fault locator	via author	[188]
JaCoCo	Java code coverage Library	open source	[236]
JavaPDG	A new platform for program dependence analysis	openly accessible	[326]
JCoverage	A tool for coverage analysis	open source	[84]
JCrasher	Java test cases generator to exhibit the error	open source	[81]
Jhawk	Java static analysis tool	Commercial	[31]
JMutator	Mutation tool using seven mutation operators for Java programs	open source	[50]
JTracor	Tool for collecting execution trace for Java programs	via author	[50]
JUMBLE	Tool for detecting destructive races	via author	[114]
Phoenix Framework	A framework for developing compilers as well as program analysis, testing and optimization	openly accessible (from Microsoft)	[74]
Microsoft Visual Studio Debugger	A debugging tool embedded in Microsoft Visual Studio	commercial	[255]
MZoltar	Automatic debugging tool for android applications	via author	[236]
Pinpoint	Fault localization tool using Jaccard coefficient	via author	[12]
Penelope	Tool for atomicity violations detection	via author	[329]
RADAR	Debugging tool for regression problems in C/C++ programs	openly accessible	[287]
RacerX	Debugging tool for concurrent program	via author	[106]
Signpost	Tool for matching program behavior against known faults	via author	[30]
SLAM toolkit	Debugging tool using static analysis	openly accessible	[44]
SLOCCount	Tool for counting executable statements	open source	[168], [223]
Spyder	Back-tracing debugger based on dynamic slicing	via author	[96]
Tarantula	Fault localization tool using Tarantula	openly accessible	[12], [70], [187], [223]
TPTP	Eclipse plugin for profiling	openly accessible	[108]
VIDA	Visual interactive debugging tool	via author	[145]
VHDLDIAG	A VHDL fault localization tool based on model-based diagnosis	via author	[376]
WhoseFault	Debugging assignment tool	via author	[323]
Whyline	An interactive debugging tool	openly accessible	[195]
Xlab	X window system events recorder	open source	[238]
Zoltar	Spectrum-based fault localization tool	via author	[170]
Zoltar-M	Tool for detecting multiple bugs	via author	[13]
χProf	Tool using execution trace to locate performance bottlenecks	via author	[22]
χRegress	Regression test set minimization tool using program coverage and execution cost	via author	[22]
χSuds	Tool for collecting execution trace information for C programs	via author	[22], [366], [367], [369], [370], [371], [372], [373], [374]
χVue	Heuristics involving the control graph, execution trace, and the maintainer's knowledge to help locate features and identify feature interactions	via author	[22]

A popular assumption is that multiple bugs in the same program perform independently. Debroy and Wong [90] examine possible interactions that may take place between different bugs, and they find that such interferences may manifest themselves to either trigger or mask some execution failures. Results based on their experiments indicate that destructive interference (when execution fails due to a bug but no longer fails when another bug is added to the same program) is more common than constructive interference (when execution fails in the presence of two bugs in the same program but does not in the presence of either bug alone) because failures are masked more often than triggered by additional bugs. It is also possible that a program with multiple bugs suffers from both destructive and constructive interferences. DiGiuseppe and Jones [102] also report that multiple bugs have an adverse impact on the effectiveness of spectrum-based techniques.

One way to debug a multiple-bug program is to follow the *one-bug-at-a-time* approach. If a program experiences some failures while it is executed against test cases of a given test suite, this approach helps programmers find and fix a bug. Then, the modified program is tested again using all the test cases in the given test suite. If any of the executions fail, additional debugging is required to find and fix the next bug. This process continues until no failure is observed. At this point, even though the program may still contain other bugs, they cannot be detected by the current suite of test cases. This approach has been adopted in studies using the DStar technique [371] and a reasoning fault localization technique based on a Bayesian reasoning framework [14]. A potential weakness of most techniques based on Bayesian reasoning (e.g., [14], [64], [193]) is that they all assume program components fail independently; in other words, interferences between multiple bugs are ignored, which is not necessarily the case in practice.

In [184], Jones et al. suggest that multiple bugs in a program can be located in parallel. The first step is to group failed test cases into different *fault-focusing* clusters such that those in the same cluster are related to the same bug. Then, the Tarantula fault localization technique [185], failed tests in each cluster, and all the successful tests are used to identify the suspicious code for the corresponding bug.

There are different ways to cluster failed test cases. One approach is to use execution profiles. Podgurski et al. [293] apply *supervised and unsupervised pattern classifications* as well as *multivariate visualization* to execution profiles of failed test cases in order to group them into fault-focusing clusters. Steimann and Frenkel [332] use the *Weil-Kettler algorithm*, a technique widely used in integer linear programming, to cluster failed test cases.

However, clustering based on the similarity between execution profiles may not reflect an accurate causation relationship between certain faults and the corresponding failed executions. For example, two failed tests, even associated with the same bug, may have very different execution profiles. It is possible for clustering techniques based on execution profiles to separate these two failed tests into different clusters.

To overcome this problem, Liu and Han [173], [226] further investigate the *due-to* relationship between failed tests and underlying bugs. They apply SOBER [172] to each failed test case and all the successful tests to generate a corresponding predicate ranking. The weighted Kendall tau distance is computed between these rankings. The distance between two rankings is small if they identify similar suspicious predicates. It also implies the rank-proximity (R-proximity) between them is high. Failed test cases with high R-proximity are clustered together, as they are likely to have the same *due-to* relationship.

Other variations include the use of more effective fault localization techniques (such as Crosstab [369], RBF [367], and DStar [371]) instead of Tarantula or SOBER, or using only a subset (refer to Section 7.2), rather than all, of the successful tests. These variations are yet to be explored.

7.2 Inputs, Outputs and Impact of Test Cases

In addition to failed and successful test cases, many (although not all) techniques discussed in Section 3 also need information about how the underlying program/model is executed with respect to each test case. Such details can be provided via different execution profiles (e.g., coverage in terms of statement, predicate, etc.).

The output of many spectrum-based (Section 3.2) fault localization techniques (such as Tarantula) is a suspiciousness ranking with statements ranked in descending order of their suspiciousness values (such as the rightmost column of Table 3). To locate a bug, programmers will examine statements at higher positions of a ranking before statements at lower positions because the former, with higher suspiciousness values, are more likely to contain bugs than the latter. On the other hand, many slice-based techniques (Section 3.1) only return a set of statements without specific ranking. Referring to Table 2, the static slice for the variable *product* is a set of eight statements, including s_1, s_2, s_4, s_5, s_7, s_8, s_{10}, and s_{13}. However, it does not tell programmers which statements are more likely to contain bugs and should therefore be examined first for possible bug locations.

Techniques discussed in Sections 3.3 (statistics-based), 3.5 (machine learning-based) and 3.6 (data mining-based) are likely[6] to generate outputs in terms of suspiciousness rankings similar to those generated by the spectrum-based techniques, whereas program state-based (Section 3.4) and model-based (Section 3.7) techniques are more likely to output a set of program/model components that will possibly contain bugs but do not explicitly specify the ranking of each component. Although both types of outputs provide suspicious components (statements, predicates, etc.) to help locate bugs, the former further prioritizes these components based on their suspiciousness values, but the latter does not.

The suite of test cases used in the program debugging is another important factor that may affect the effectiveness of a fault localization technique. Some fault localization techniques (e.g., [21], [77], [133], [134], [140], [303], [400]) focus on locating program bugs using either a single failed test case or a single failed test case with a few successful test cases. Others (e.g., [185], [222], [223], [366], [367], [369], [373], [374], [375]) use multiple failed and successful test cases. These latter techniques take advantage of more test cases

[6] Since there are many techniques in each category, it is possible that a particular technique may behave differently from others in the same category in terms of which types of output are generated.

than the former, so it is likely that the latter are more effective in locating program bugs. For example, Tarantula [185] which uses multiple failed and multiple successful tests, has been shown to be more effective than nearest neighbor [303], a technique that only uses one failed and one 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 [77] when compared to the techniques that take into account multiple successful and failed test cases simultaneously.

Although techniques using multiple failed and multiple successful test cases may have better fault localization effectiveness, an underlying assumption is that a large set of such tests is available. This may also lead to the assumption of existence of an oracle that can be used to automatically determine whether an execution is successful or failed. Unfortunately, this may not be true in the real world, as a test oracle can be incomplete, out-of-date, or ambiguous. Studies such as [160], [161] have reported that for many systems and for much of testing as currently practiced in industry, testers do not have formal specifications, assertions, or automated oracles. As a result, they face the potentially daunting task of manually checking the system's behavior for all test cases executed. In response to this challenge, researchers have presented various solutions [16], [147], [148]. Nevertheless, how to generate an automated test oracle still remains an issue that needs to be further explored. Hence, we cannot take it for granted that there are multiple tests with all execution results (success or failure) known.

Using a test suite that does not achieve high coverage of the target program may have an adverse impact on the fault localization results. During test generation, different criteria (e.g., requirements-based boundary value analysis, or white-box-based statement or decision coverage) can be used as guidance. Diaz et al. [100] use a meta-heuristic technique (a so-called Tabu Search approach) to automatically generate a test suite to obtain maximum branch coverage. In [33], [34], [35], Artzi et al. present a tool called Apollo to generate test cases automatically based on combined concrete and symbolic executions. Apollo first executes a program on an empty input and records a path constraint that reflects the program's executed control-flow predicates. New inputs are then generated by changing predicates in the path constraint and solving the resulting constraints. Executing the program on these inputs produces additional control-flow paths. Failures observed during executions are recorded. This process is repeated until a pre-defined threshold of statements coverage is reached, a sufficient number of faults are detected, or the time budget is exhausted. Authors of [176] suggest that test suites satisfying branch coverage are better than those satisfying statement coverage in effectively supporting fault localization, whereas authors of [177] claim that test suites satisfying MC/DC coverage are better than those satisfying branch coverage.

Furthermore, in [320], Santelices et al. study the fault localization effectiveness of Tarantula using three types of program coverage—statements, branches, and define-use pair. They conclude that Tarantula using define-use pair coverage is more effective and stable than that using branch coverage, which is more effective than that using statement coverage. Based on this, the authors further propose to use a combination of the three types of coverage to achieve better fault localization effectiveness.

Some researchers argue that it is not efficient to use all the test cases in a given test suite to locate program bugs. Instead, they use either *test case reduction* by selecting only a subset of test cases or *test case prioritization* by assigning different priorities to different cases to improve the efficiency of fault localization techniques [45], [48], [53], [62], [63], [122], [126], [127], [128], [146], [175], [176], [348], [362], [399]. One approach of test prioritization is to give higher priority to failed test cases that execute fewer statements, as they provide more information and minimize the search domain [263]. In [119], the authors propose an approach to generate balanced test suites in order to improve fault localization effectiveness by cloning failed test cases a suitable number of times to match the number of successful test cases. Rößler et al. [307] propose a technique, BUGEX, which applies dynamic symbolic execution to generate test cases with a minimal difference from the execution path of a single failed test case. Based on the generated test cases, the branches that are executed by more failed test cases but fewer successful test cases are more likely to cause the failure. The study in [179] applies a similar test case generation approach, but the generated test cases are instead used with a spectrum-based fault localization technique to rank basic blocks in descending order according to their suspiciousness values.

Baudry et al. [50] use a bacteriological approach (which is an adaptation of genetic algorithms) to bridge the gap between testing and diagnosis (fault localization) based on a *test-for-diagnosis* criterion. Test cases are generated to satisfy this criterion so that diagnosis algorithms can be used more efficiently. Their objective is to achieve a better diagnosis (a more efficient fault localization) using a minimal number of test cases. Studies such as [127], [253] focus on a cross evaluation of the impact of different test reduction and prioritization techniques on the efficiency of software fault localization.

Test execution sequence also has an impact on program debugging. For example, it is possible that a program execution fails not because of the current test but because of a previous test that does not set up an appropriate execution environment for the current test. If a failure cannot be observed unless a group of test cases are executed in a specific sequence, then these test cases should be bundled together as one single failed test.

7.3 Coincidental Correctness

The concept of *coincidental correctness*, introduced by Budd and Angluin in [59], discusses 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. This phenomenon can occur for many reasons. For example, given a faulty statement in which a variable is assigned with an incorrect value, in one test execution, this value may affect the output of the program and result in a failure. However, in another test execution, the value of this variable is later overwritten. Thus, the output of the program is not affected and failure is not triggered. Studies discussing *coincidental correctness* have been reported in recent years [44], [46], [159], [218], [239], [254], [355], [419].

Coincidental correctness can negatively impact the effectiveness of fault localization techniques. Ball et al. [44] claim that this is the reason why their technique fails to locate bugs in three out of 15 single-bug programs. Wang et al. [355] conclude that the effectiveness of Tarantula decreases when the frequency of *coincidental correctness* is high and increases when the frequency is low.

To overcome this problem, Masri and Assi [239] propose a technique to clean test suites by removing test cases that may introduce possible coincidental correctness for better fault localization effectiveness. Their technique is further enhanced by using fuzzy test suites and clustering analysis [240]. Bandyopadhyay and Ghosh [46] suggest a different approach by first measuring the likelihood of coincidental correctness of a successful test case based on the average proximity of its execution profile with that of all failed test cases. Such likelihood is assigned as the weight of the corresponding successful test case and used for subsequent suspiciousness computation. Zhang et al. [419] present FOnly, a technique that relies only on failed test cases to locate bugs statistically, even though fault localization commonly relies on both successful and failed tests. Authors of [418] propose a fault localization technique, BlockRank, to calculate, contrast, and propagate the *mean edge profiles* between successful and failed executions to alleviate the impact of coincidental correctness.

7.4 Faults Introduced by Missing Code

One claim that can generally be made against fault localization techniques discussed in this survey is that they are incapable of locating bugs resulting from missing code. For example, slice-based techniques will never be able to locate such bugs – since the *faulty* code is not even in the program. Therefore, this code will not appear in any of the slices. Based on this, one might conclude that most fault localization techniques are inappropriate for locating such bugs. Although 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 trigger some adverse effects elsewhere in the program execution, 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 hint to programmers that some omitted statements may be leading to control-flow anomalies. This implies that we are still able to identify suspicious code related to the omission error, such as the affected decision branch using slice-based techniques. A similar argument can also be made for other techniques, including but not limited to program spectrum-based (Section 3.2), statistics-based (Section 3.3), and program state-based techniques (Section 3.4). Thus, even though software fault localization techniques may not be able to pinpoint the exact locations of missing code, they can still provide a good starting point for the search.

7.5 Combination of Multiple Fault Localization Techniques

The effectiveness of a fault localization technique is very much scenario dependent, affected by successful and failed test cases, program structures and semantics, nature of the bugs, etc. There is no single technique superior to all others in every scenario. Thus, it makes sense to combine multiple techniques and retain the good qualities of individual techniques while mitigating the drawbacks of each. In [91], [92], Debroy et al. propose a way to do so by combining the rankings of statements generated by multiple techniques. The advantage of this approach (i.e., combining the rankings) over a design-based integration approach (in which the actual techniques would somehow be incorporated to form a new technique) is that it is more cost-effective to realize and is always extensible. Based on a similar idea, Lucia and Xia [232] use two normalization methods to combine results of different fault localization techniques.

In [9], Abreu et al. address the inherent limitations of spectrum-based fault localization techniques, stating that component semantics of the program are not considered. They propose a way to enhance the diagnostic quality of a spectrum-based fault localization technique by combining it with a model-based debugging approach using the abstraction interpretation generated by a framework called DEPUTO. More precisely, a model-based approach is used to refine the ranking via filtering to exclude those components that do not explain the observed failures when the program's semantics are considered.

In [350], Wang et al. use two different search algorithms, simulated annealing and genetic algorithm, to find approximate optimal compositions from 22 existing spectrum-based fault localization techniques. However, a search-based approach lacks flexibility and efficiency. For flexibility, the search must be re-performed to update the optimal composition whenever a new fault localization technique is included. Also, an optimal composition for one program may not be the optimal for another program, which means the search process needs to be re-performed when the subject program changes. For efficiency, the potential large size of search space makes the search process very time consuming.

Spectrum-based and slice-based techniques are both widely used. Combinations between techniques from these two categories have been reported [28], [165], [214], [363]. For example, in [28], Alves et al. combine Tarantula and dynamic slicing to improve fault localization effectiveness. First, all the statements in a program are ranked based on their suspiciousness calculated by using the Tarantula technique. Then, a dynamic slice with respect to a failure-indicating variable at the failure point is generated. Statements not in this slice will be removed from the ranking to further reduce the search domain. In [188], Ju et al. propose a hybrid slice-based fault localization technique combining dynamic and execution slices. A prototype tool, hybrid slice spectrum fault locator (*HSFal*), is implemented to support this technique.

Hofer and Wotawa [165] emphasize that spectrum-based fault localization techniques (e.g., Ochiai [12]) operated at a basic block level do not provide fine-grained results, whereas techniques based on slicing-hitting-set-computation (e.g., the HS-Slice algorithm [379]) sometimes produce an undesirable ranking with statements (such as constructors), which are executed by many test cases, at the top. To eliminate these drawbacks, techniques of these two types should be combined.

Other combinations have also been explored. In [33], Artzi et al. combine Tarantula and a technique for output mapping to reduce the number of statements that need to be examined. A similar approach is repeated in which Tarantula is replaced by Ochiai and Jaccard [34]. In [129], Gopinath et al. apply spectrum-based localization in synergy with specification-based analysis to more accurately locate bugs. The key idea is that unsatisfiability analysis of violated specifications, enabled by SAT technology, can be used to compute unsatisfiable cores, including statements that are likely to contain bugs. In [61], Burger and Zeller propose a technique, JINSI, which combines delta debugging and dynamic slicing for effective fault localization. JINSI takes a single failed execution and treats it as a series of object interactions (e.g., method calls and returns) that eventually produce the failure. The number of interactions will be reduced to the minimum number required to reproduce the failure, which will reduce the search space needed to locate the corresponding bug.

7.6 Ties within Fault Localization Rankings

As discussed earlier (referring to Section 3.2), statements with the same suspiciousness are tied for the same position in a ranking. Results of a study by Xu et al. [393], using three fault localization techniques on four sets of programs, show that the symptom of assigning the same suspiciousness to multiple statements (i.e., the existence of ties in a produced ranking) appears everywhere and is not limited to any particular technique or program. Under such a scenario, the total number of statements that a programmer needs to examine in order to find the bugs may vary considerably. In response, two levels of effectiveness, the *best* and the *worst*, are computed (see Section 5: Evaluation Metrics). In practice, the more the ties, the bigger the difference between the best and the worst effectiveness. Ties also make the exact effectiveness of a fault localization technique more uncertain.

In voting scenarios when voters are unable to select between two or more alternatives, the candidates are ranked based on some key or natural ordering, such as an alphabetical ordering, to break ties. Similarly, when two statements are tied for the same ranking, the line numbers assigned to them in a text editor can serve as the key. Other techniques such as confidence-based strategy and data dependency-based strategy are also used to break ties [369], [366], [386], [393].

7.7 Fault Localization for Concurrency Bugs

Concurrent programs suffer most from three kinds of access anomalies: data race [32], [321], atomicity violation [110], [113], [115], and atomic-set serializability violations [24], [44].

Among the approaches that have mushroomed in recent years, predictive analysis-based techniques haven drawn significant attention [111], [113], [115], [116], [322], [349]. Generally speaking, these techniques record a trace of program execution, statically generate other permutations of these events, and expose unexercised concurrency bugs. One potential problem of these techniques is that they may sometimes report a large number of false positives. For example, only six of 97 reported atomicity violations in a study using Atomizer (a dynamic atomicity checker) are real [116]. On the contrary, a study in [329] using a different tool, Penelope, for atomicity violations detection reports no false positive.

Tools such as Chord [262] and RacerX [106] can statically analyze a program to find concurrency bugs. However, since all paths need to be explored, it is impractical to apply these tools to large, complicated programs. A runtime analysis (such as [144], [321], [392]), on the other hand, is less powerful than a static analysis but also produces fewer false alarms. The drawback is that only faults manifested in some specific executions can be detected.

Another approach for bug localization in concurrent programs is to use model checking [60], [190], [261], [324]. For instance, Shacham et al. [324] use a model checker to construct the evidence for data race reported by the lockset algorithm. However, due to the possible exponential size of the search space, it is difficult to adopt this approach for large-sized programs without compromising its detection capability.

There are other techniques for detecting concurrency bugs. For example, Flanagan and Freund use a prototype tool JUMBLE to explore the non-determinism of relaxed memory models and to detect destructive races in the program [114]. Park et al. apply a CTrigger testing framework [282] to detect real atomicity violations by controlling the program execution to exercise low-probability thread interleavings. Park also presents a study to debug non-deadlock concurrency bugs [283]. Wang et al. [353] propose a technique to locate buggy shared memory accesses that are responsible for triggering concurrency bugs. Authors of [345] propose a tool, MEMSAT, to help in debugging memory models. Koca et al. [198] locate faults in concurrency programs using an idea similar to spectrum-based fault localization techniques.

7.8 Spreadsheet Fault Localization

Spreadsheet systems represent a landmark in the history of generic software products. It is estimated that 95 percent of all U.S. firms use spreadsheets for financial reporting [280], 90 percent of all analysts in the industry perform calculations in spreadsheets [280] and 50 percent of all spreadsheets are the basis for decisions [156]. Such wide usage, however, has not been accompanied by effective mechanisms for bug prevention and detection, as shown by studies such as [278], [281]. As a result, bugs in spreadsheets are to be blamed for a long list of real problems compiled and available at the European Spreadsheet Risk Interest Group's (EuSpRIG) web site (http://www.eusprig.org/). A recent study by Reinhart and Rogoff [300] also gives a similar conclusion. In response to this, many studies regarding spreadsheet fault localization have been reported [1], [3], [23], [56], [157], [158], [162], [164], [169], [311], [314].

A model-based spreadsheet fault localization technique is presented in [169], using an extended hitting-set algorithm and user-specified or historical test cases and assertions to identify possible error causes. Hofer et al. [164], apply a constraint-based representation of spreadsheets and a general constraint solver to locate bugs in spreadsheets. Another constraint-based approach for debugging faulty spreadsheets (CONBUG) is presented by Abreu et al. [15], taking a spreadsheet and one test case as input to compute a set of faulty candidates. Abraham and Erwig [2] describe a tool, GoalDebug, for debugging spreadsheets, using a constraint-based approach similar to that in [164]. Whenever the computed output of a cell is incorrect, users can provide an expected value, which is employed to produce a list of possible changes to the corresponding formulae that, when applied, will generate the user-specified output. This involves mutating the spreadsheet based on a set of predefined change (repair) rules and ascertaining whether user expectations are met. A similar approach also appears in other studies such as [95] and [152]. Authors of [95] propose a strategy for automatically fixing bugs in both Java and C programs by combining mutation testing and software fault localization. An approach of using path-based weakest preconditions is discussed in [152] to generate program modifications for bug fixing.

Abraham and Erwig also present a system, UCheck, which infers header information in spreadsheets, performs a unit analysis, and notifies users when bugs are detected [3]. Hermans et al. [157] suggest a way to locate spreadsheet smells (possible weak points in the spreadsheet design) and display them to users in data-flow diagrams. An approach to detect and visualize data clones (caused by copying the value computed by a formula in one cell as plain text to a different cell) in spreadsheets is reported in [158].

Other techniques aimed at reducing the occurrence of errors in spreadsheets include code inspection [279], refactoring [42], and adoption of better spreadsheet design practices [82], [83].

7.9 Theoretical Studies

Instead of being evaluated empirically, the effectiveness of software fault localization techniques can also be analyzed from theoretical perspectives.

Briand et al. [57] report that the formula used to compute the suspiciousness of a given statement by Tarantula can be re-expressed so that the suspiciousness only depends on the ratio of the number of failed tests (N_{CF}) to the number of successful tests (N_{CS}) that execute the statement. Lee et al. [182], [212] prove that Tarantula always produces a ranking identical to that of a technique where the suspiciousness function is formulated as $\frac{N_{CF}}{N_{CF}+N_{CS}}$. A study by Naish et al. [267] examines over 30 formulae and divides them into groups such that those in the same group are equivalent for ranking. Independently, Debroy and Wong [93] also report a similar study showing that some similarity coefficient-based fault localization techniques are equivalent to one another.

Xie et al. [384] perform a theoretical study on the effectiveness of some spectrum-based fault localization techniques. Based on the *risk values* (which is the same as *suspiciousness* discussed in this survey), program statements are assigned to one of the three sets, S_B^R, S_F^R, and S_A^R, based on whether their risk values are higher than, the same as, or lower than the value of the statement containing the bug. The authors make three assumptions: i) a faulty program has exactly one fault; ii) for any given single-fault program, there is exactly one faulty statement; and iii) this faulty statement must be executed by all failed tests. They also assume that the underlying test suite must have 100 percent statement coverage. Unfortunately, many of these assumptions are over-simplified and do not hold for real-life programs. With respect to some selected techniques (many of which are similarity coefficient-based), they examine the *subset* relation between S_B^R and S_A^R generated by the corresponding ranking formulae and conclude that for two techniques, R_1 and R_2, if $S_B^{R_1} \subseteq S_B^{R_2}$ and $S_A^{R_2} \subseteq S_A^{R_1}$ then R_1 is *better* (more *effective*) than R_2 such that the number of statements examined by R_1 is less than that examined by R_2 to find the first faulty statement. One problem of this proof as reported in [371] is that it does not consider statements in S_F^R. As a result, for some special cases, even though the proof indicates that one technique is more effective than another, the former has to examine more statements than or the same number of statements as the latter $-$ contradicting the result of the proof. Another weakness is that some advanced and more effective techniques (e.g., [14], [223], [367], [371]) are excluded, even though they use exactly the same input data as those included in [384]. Authors of [210] also question the validity of [384]. They compare the effectiveness of the five *best* fault localization techniques based on the theoretical study in [384] with the effectiveness of Tarantula and Ochiai, and they find that the latter are significantly more effective than the former. This directly contradicts the conclusion of [384]. Xie et al. [388] also apply their theoretical analysis framework to 30 genetic programming-evolved formulae and show that some of them can be used for fault localization. However, they make the same over-simplified assumptions as those in [384].

There are other theoretical studies for single-bug programs. For example, Lee et al. [265] identify a class of *strictly rational* fault localization techniques in which the suspicious value of a statement strictly increases if this statement is executed by more failed test cases and strictly decreases if this statement is executed by more successful test cases. The authors claim that strictly rational techniques do not necessarily outperform those that are not. Therefore, limited attention should be given to these strictly rational techniques. In [264], Lee et al. further identify a class of *optimal* fault localization techniques for locating *deterministic* bugs (similar to Bohrbugs defined in [139]) that will always cause test cases to fail whenever they are executed.

8 Conclusion

As today's software has become larger and more complex than ever before, software fault localization accordingly requires a greater investment of time and resources. Consequently, locating program bugs is no longer an easily-automated mechanical process. In practice, locations based on intelligent guesses of experienced programmers with expert knowledge of the software being debugged should be examined first. However, if this fails, an appropriate fallback would be to use a systematic technique (such as those discussed in this survey) based on solid reasoning and supported by case studies, rather than to use an unsubstantiated ad hoc approach. This is why techniques that can help programmers effectively locate bugs are highly in demand, which also stimulates the proposal of many fault localization techniques from a widespread perspective. It is imperative that software engineers involved with developing reliable and dependable systems have a good understanding of existing techniques, as well as an awareness of emerging trends and developments in the area. To facilitate this, we conduct a detailed survey and present the results so that software engineers at all program debugging experience levels can quickly gain necessary background knowledge and the ability to apply cost-effective software fault localization techniques tailored to their specific environments.

In this survey, a publication repository has been created, including 331 papers and 54 Ph.D. and Masters' theses on software fault localization from 1977 to November 2014. These techniques are classified into eight categories: slice-based, spectrum-based, statistics-based, program state-based, machine learning-based, data mining-based, model-based, and miscellaneous. The figures and tables presented in the previous sections strongly indicate that software fault localization has become an important research topic on the front burner and suggest the trend of ongoing research directions.

Our analysis shows that the numbers of published papers in each category differ from each other and that the research interest shifts from one category to another as time moves on. For example, static and dynamic slice-based techniques were popular between 2004 and 2007, whereas execution slice and program spectrum-based techniques have dominated since 2008.

Different metrics to evaluate the effectiveness of software fault localization techniques (in terms of how much code needs to be examined before the first faulty location is identified) are reviewed, including T-score, EXAM score/*Expense*, P-score, N-score, and Wilcoxon signed-rank test. Subject programs and debugging tools used in various empirical evaluations are summarized. Results of different empirical studies using these metrics, programs, and tools suggest that no one category is completely superior to another. In fact, techniques in each category have their own advantages and disadvantages.

Additionally, effectiveness of these techniques can also be analyzed from theoretical perspectives. However, such analyses very often make over-simplified and non-realistic assumptions that do not hold for real-life programs. Hence, their conclusions in general are only applicable within limited scopes. This implies that a theoretical analysis alone is not enough. It is advisable to apply both empirical evaluations and theoretical analyses to provide a more complete assessment.

We emphasize that effectiveness is not the only attribute of a software fault localization technique that should be considered. Other factors, including overhead for computing the suspiciousness of each program component, time and space for data collection, human effort, and tool support, should be included as well. We also discuss aspects that are critical to software fault localization, such as fault localization on programs with multiple bugs, concurrent programs, and spreadsheets, as well as impacts of test cases, coincidental correctness, and faults introduced by missing code.

To conclude, our objective is to use this survey to provide the software engineering community with a better understanding of state-of-the-art research in software fault localization and to identify potential drawbacks and deficiencies of existing techniques so that additional studies can be conducted to improve their practicality and robustness.

Acknowledgments

W. Eric Wong is the corresponding author of this paper.

References

[1] R. Abraham and M. Erwig, "Header and unit inference for spreadsheets through spatial analyses," in *Proc. IEEE Symp. Visual Languages Human Centric Comput.*, Roma, Italy, Sep. 2004, pp. 165–172.

[2] R. Abraham and M. Erwig, "Goaldebug: A spreadsheet debugger for end users," in *Proc. IEEE Int. Conf. Softw. Eng.*, Minneaplis, MN, USA, May 2007, pp. 251–260.

[3] R. Abraham and M. Erwig, "Ucheck: A spreadsheet type checker for end users," *J. Visual Languages Comput.*, vol. 18, no. 1, pp. 71–95, Feb. 2007.

[4] D. Abramson, I. Foster, J. Michalakes, and R. Sosic, "Relative debugging and its application to the development of large numerical models," presented at the *8th Int. Conf. High Perform. Comput., Netw., Storage, Anal.*, no. 51, San Diego, CA, USA, Dec. 1995

[5] R. Abreu, "Spectrum-based fault localization in embedded software," Ph.D. dissertation, Univ. Minho, Braga, Portugal, 2009

[6] R. Abreu and A. J. C. van Gemund, "A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis," in *Proc. 8th Symp. Abstraction, Reformulation, Approximation*, Lake Arrowhead, CA, USA, Jul. 2009, pp. 1–8.

[7] R. Abreu, A. González, and A. J. Gemund, "Exploiting count spectra for Bayesian fault localization," presented at the *6th Int. Conf. Predictive Models Softw. Eng.*, Timisoara, Romania, Sep. 2010

[8] R. Abreu, A. González, P. Zoeteweij, and A. J. van Gemund, "Automatic software fault localization using generic program invariants," in *Proc. ACM Symp. Appl. Comput.*, Ceara, Brazil, Mar. 2008, pp. 712–717.

[9] R. Abreu, W. Mayer, M. Stumptner, and A. J. van Gemund, "Refining spectrum-based fault localization rankings," in *Proc. ACM Symp. Appl. Comput.*, Honolulu, HI, USA, Mar. 2009, pp. 409–414.

[10] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. van Gemund, "A practical evaluation of spectrum-based fault localization," *J. Syst. Softw.*, vol. 82, no. 11, pp. 1780–1792, Nov. 2009.

[11] R. Abreu, P. Zoeteweij, and A. J. van Gemund, "An evaluation of similarity coefficients for software fault localization," in *Proc. Pacific RIM Int. Symp. Dependable Comput.*, Riverside, CA, USA, Dec. 2006, pp. 39–46.

[12] R. Abreu, P. Zoeteweij, and A. J. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proc. Testing: Academic Ind. Conf. Practice Res. Techn.*, Windsor, CT, USA, Sep. 2007, pp. 89–98.

[13] R. Abreu, P. Zoeteweij, and A. J. Gemund, "Localizing software faults simultaneously," in *Proc. 9th Int. Conf. Quality Softw.*, Jeju, Korea, Aug. 2009, pp. 367–376.

[14] R. Abreu, P. Zoeteweij, and A. J. van Gemund, "Spectrum-based multiple fault localization," in *Proc. 24th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Auckland, CA, USA, Nov. 2009, pp. 88–99.

[15] R. Abreu, B. Hofer, A. Perez, and F. Wotawa, "Using constraints to diagnose faulty spreadsheets," *Softw. Quality J.*, vol. 23, pp. 297–322, May 2014.

[16] S. Afshan, P. McMinn, and M. Stevenson, "Evolving readable string test inputs using a natural language model to reduce human oracle cost," in *Proc. IEEE 6th Int. Conf. Softw. Testing, Verification Validation*, Mar. 2013, pp. 352–361.

[17] H. Agrawal, "Towards automatic debugging of computer program," Ph.D. dissertation, Purdue Univ., West Lafayette, IN, USA, 1991.

[18] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Debugging with dynamic slicing and backtracking," *Softw.–Practice Experience*, vol. 23, no. 6, pp. 589–616, Jun. 1993.

[19] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "An execution backtracking approach to program debugging," *IEEE Softw.*, vol. 8, no. 5, pp. 21–26, May 1991.

[20] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, White Plains, NY, USA, Jun. 1990, pp. 246–256.

[21] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," in *Proc. 6th Int. Symp. Softw. Rel. Eng.*, Toulouse, France, Oct. 1995, pp. 143–151.

[22] H. Agrawal, J. R. Horgan, J. J. Li, S. London, W. E. Wong, S. Ghosh, and N. Wilde, "Mining system tests to aid software maintenance," *IEEE Comput.*, vol. 31, no. 7, pp. 64–73, Jul. 1998.

[23] Y. Ahmad, T. Antoniu, S. Goldwater, and S. Krishnamurthi, "A type system for statically detecting spreadsheet errors," in *Proc. IEEE Int. Symp. Autom. Softw. Eng.*, Montreal, QC, Canada, Oct. 2003, pp. 174–183.

[24] Z. A. Al-Khanjari, M. R. Woodward, H. A. Ramadhan, and N. S. Kutti, "The efficiency of critical slicing in fault localization," *J. Softw. Quality Control*, vol. 13, no. 2, pp. 129–153, Jun. 2005.

[25] S. Ali, J. H. Andrews, T. Dhandapani, and W. Wang, "Evaluating the accuracy of fault localization techniques," in *Proc. IEEE Int. Symp. Autom. Softw. Eng.*, Auckland, New Zealand, 2009, pp. 76–87.

[26] S. Ali, "Localizing state-dependent faults using associated sequence mining," Ph.D. dissertation, Univ. Western Ontario, London, ON, Canada, 2013.

[27] M. A. Alipour and A. Groce, "Extended program invariants: Applications in testing and fault localization," in *Proc. 9th Int. Workshop Dyn. Anal.*, Minneapolis, MN, USA, Jul. 2012, pp. 7–11.

[28] E. Alves, M. Gligoric, V. Jagannath, and M. d'Amorim, "Fault-localization using dynamic slicing and change impact analysis," in *Proc. IEEE Int. Symp. Autom. Softw. Eng.*, Lawrence, KS, USA, Nov. 2011, pp. 520–523.

[29] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proc. 27th Int. Conf. Softw. Eng.*, St. Louis, Missouri, USA, May 2005, pp. 402–411.

[30] M. Andrews, "Signpost: Matching program behavior against known faults," *IEEE Softw.*, vol. 20, no. 6, pp. 84–89, Nov./Dec. 2003.

[31] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *J. Syst. Softw.*, vol. 83, no. 1, pp.2–17, Jan. 2010.

[32] C. Artho, K. Havelund, and A. Biere, "High-level data races," *J. Softw. Testing, Verification Rel.*, vol. 13, pp. 207–227, 2003.

[33] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Practical fault localization for dynamic web applications," in *Proc. IEEE Int. Conf. Softw. Eng.*, Cape Town, South Africa, May 2010, pp. 265–274.

[34] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, "Finding bugs in web applications using dynamic test generation and explicit-state model checking," *IEEE Trans. Softw. Eng.*, vol. 36, no. 4, pp. 474–494, Jul./Aug. 2010.

[35] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization," in *Proc. IEEE Int. Symp. Softw. Testing Anal.*, Trento, Italy, Jul. 2010, pp. 49–60.

[36] L. C. Ascari, L. Y. Araki, A. R. T. Pozo, and S. R. Vergilio, "Exploring machine learning techniques for fault localization," in *Proc. 10th Latin Am. Test Workshop*, Buzios, Brazil, Mar. 2009, pp. 1–6.

[37] A. Avizienis, J. C. Laprie, B. Randell, and C. E. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secure Comput.*, vol. 1, no. 1, pp. 11–33, Jan.–Mar. 2004.

[38] G. K. Baah, A. Podgurski, and M. J. Harrold, "Causal inference for statistical fault localization," in *Proc. Int. Symp. Softw. Testing Anal.*, Trento, Italy, Jul. 2010, pp. 73–83.

[39] G. K. Baah, A. Podgurski, and M. J. Harrold, "Mitigating the confounding effects of program dependences for effective fault localization," in *Proc. Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, Szeged, Hungary, Sep. 2011, pp. 146–156.

[40] G. K. Baah, A. Podgurski, and M. J. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis," *IEEE Trans. Softw. Eng.*, vol. 36, no. 4, pp. 528–545, Jul./Aug. 2010.

[41] G. K. Baah, "Statistical causal analysis for fault localization," Ph.D. dissertation, Georgia Inst. Technol., Atlanta, GA, USA, 2012

[42] S. Badame and D. Dig, "Refactoring meets spreadsheet formulas," in *Proc. Int. Conf. Softw. Maintenance*, Trento, Italy, Sep. 2012, pp. 399–409.

[43] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," *ACM Trans. Program. Languages Syst.*, vol. 16, no. 4, Jul. 1994.

[44] T. Ball, M. Naik, and S. K. Rajamani, "From symptom to cause: Localizing errors in counterexample traces," in *Proc. ACM SIGPLAN-SIGACT Symp. Principles Program. Languages*, Venice, Italy, Jan. 2003, pp. 97–105.

[45] A. Bandyopadhyay, "Improving spectrum-based fault localization using proximity-based weighting of test cases," in *Proc. IEEE Int. Symp. Autom. Softw. Eng.*, Lawrence, KS, USA, Nov. 2011, pp. 660–664.

[46] A. Bandyopadhyay, "Mitigating the effect of coincidental correctness in spectrum based fault localization," in *Proc. Int. Conf. Softw. Testing, Verification, Validation*, Montreal, QC, Canada, Apr. 2012, pp. 479–482.

[47] A. Bandyopadhyay, "Mitigating the effect of coincidental correctness in spectrum based fault localization," Ph.D. dissertation, Colorado State Univ., Fort Collins, CO, USA, 2013.

[48] A. Bandyopadhyay and S. Ghosh, "Proximity based weighting of test cases to improve spectrum based fault localization," in *Proc. IEEE Int. Symp. Autom. Softw. Eng.*, Lawrence, KS, USA, Nov. 2011, pp. 420–423.

[49] A. Bandyopadhyay and S. Ghosh, "On the effectiveness of the tarantula fault localization technique for different fault classes," in *Proc. IEEE Int. Symp. High-Assurance Syst. Eng.*, Boca Raton, FL, USA, Nov. 2011, pp. 317–324.

[50] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving test suites for efficient fault localization," in *Proc. IEEE Int. Conf. Softw. Eng.*, Shanghai, China, May 2006, pp. 82–91.

[51] A. Beszedes, T. Gergely, Z. Szabo, J. Csirik, and T. Gyimothy, "Dynamic slicing method for maintenance of large C programs," in *Proc. Eur. Conf. Softw. Maintenance Reeng.*, Lisbon, Portugal, Mar. 2001, pp. 105–113.

[52] D. Binkley and M. Harman, "A survey of empirical results on program slicing," *Adv. Comput.*, vol. 62, pp. 105–178, Jan. 2004.

[53] J. Bo, Z. Zhang, T. H. Tse, and T. Y. Chen, "How well do test case prioritization techniques support statistical fault localization," in *Proc. Annu. IEEE Comput. Softw. Appl. Conf.*, Seattle, WA, USA, Jul. 2009, pp. 99–106.

[54] J. Bohnet, S. Voigt, and J. Döllner, "Projecting code changes onto execution traces to support localization of recently introduced bugs," in *Proc. ACM Symp. Appl. Comput.*, Mar. 2009, pp. 438–442.

[55] F. Bourdoncle, "Abstract debugging of higher-order imperative languages," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, Albuquerque, NM, USA, Jun. 1993, pp. 46–55.

[56] A. Bregar, "Complexity metrics for spreadsheet models," in *Proc. EuSpRIG*, 2004, vol. 3895, pp. 85–93.

[57] L. C. Briand, Y. Labiche, and X. Liu, "Using machine learning to support debugging with tarantula," in *Proc. IEEE Int. Symp. Softw. Rel.*, Trolhattan, Sweden, Nov. 2007, pp. 137–146.

[58] Y. Brun and M. D. Ernst, "Finding latent code errors via machine learning over program executions," in *Proc. IEEE Int. Conf. Softw. Eng.*, Edinburgh, U.K., May 2004, pp. 480–490.

[59] T. A. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Infomatica*, vol. 18, no. 1, pp. 31–45, 1982.

[60] S. Burckhardt, R. Alur, and M. M. K. Martin, "Checkfence: Checking consistency of concurrent data types on relaxed memory models," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, San Diego, USA, Jun. 2007, pp. 12–21.

[61] M. Burger and A. Zeller, "Minimizing reproduction of software failures," in *Proc. Int. Symp. Softw. Testing Anal.*, Toronto, ON, Canada, Jul. 2011, pp. 221–231.

[62] K. Cai, T. Jing, and C. Bai, "Partition testing with dynamic partitioning," in *Proc. Annu. Int. Comput. Softw. Appl. Conf.*, Scotland, U.K., Jul. 2005, pp. 113–116.

[63] J. Campos, R. Abreu, G. Fraser, and M. Amorim, "Entropy-based test generation for improved fault localization," in *Proc. 28th Int. Conf. Autom. Softw. Eng.*, Silicon Valley, CA, USA, Nov. 2013, pp. 257–267.

[64] N. Cardoso and R. Abreu, "A kernel density estimate-based approach to component goodness modeling," in *Proc. 27th AAAI Conf. Artif. Intell.*, Washington, D.C., USA, Jul. 2013, pp. 152–158.

[65] P. Cellier, M. Ducasse, S. Ferre, and O. Ridoux, "Formal concept analysis enhances fault localization in software," in *Proc. Int. Conf. Formal Concept Anal.*, Montreal, QC, Canada, Feb. 2008, pp. 273–288.

[66] P. Cellier, M. Ducasse, S. Ferre, and O. Ridoux, "Multiple fault localization with data mining," in *Proc. Int. Conf. Softw. Eng. Knowl. Eng.*, Miami, FL, USA, Jul. 2011, pp. 238–243.

[67] S. Chaki, A. Groce, and O. Strichman, "Explaining abstract counterexamples," presented at the *Int. Symp. Found. Softw. Eng.*, Newport Beach, CA, USA, Oct. 2004.

[68] C. Chen, "Automated fault localization for service-oriented software systems," Ph.D. dissertation, Delft Univ. Technol., Delft, the Netherlands, 2015.

[69] K. Chen and J. Chen, "Aspect-based instrumentation for locating memory leaks in Java programs," in *Proc. Annu. Int. Comput. Softw. Appl. Conf.*, Beijing, China, Jul. 2007, pp. 23–28.

[70] I. Chen, C. Yang, T. Lu, and H. Jaygarl, "Implicit social network model for predicting and tracking the location of faults," in *Proc. Annu. Int. Comput. Softw. Appl. Conf.*, Turku, Finland, Aug. 2008, pp. 136–143.

[71] T. Y. Chen, T. H. Tse, and Z. Q. Zhou, "Semi-proving: An integrated method for program proving, testing, and debugging," *IEEE Trans. Softw. Eng.*, vol. 37, no. 1, pp. 109–125, Jan./Feb. 2011.

[72] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan, "Identifying bug signatures using discriminative graph mining," in *Proc. Int. Symp. Softw. Testing Anal.*, Chicago, IL, USA, Jul. 2009, pp. 141–152.

[73] O. C. Chesley, "CRISP-A fault localization tool for java programs," Master thesis, The State Univ. New Jersey, New Wark, NJ, USA, 2007.

[74] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "HOLMES: Effective statistical debugging via efficient path profiling," in *Proc. IEEE Int. Conf. Softw. Eng.*, Vancouver, BC, Canada, May 2009, pp. 34–44.

[75] S. Choi, S. Cha, and C. C. Tappert, "A survey of binary similarity and distance measures," *J. Systemics, Cybern. Informat.*, vol. 8, no. 1, pp. 43–48, Jan. 2010.

[76] J. Christ, E. Ermis, M. Schaf, and T. Wies, "Flow-sensitive fault localization," in *Proc. Int. Conf. Verification, Model Checking, Abstract Interpretation*, Rome, Italy, Jan. 2013, pp. 189–208.

[77] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proc. IEEE Int. Conf. Softw. Eng.*, pp. 342–351, May 2005.

[78] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. IEEE Int. Symp. Principles Program. Languages*, Los Angeles, CA, USA, Jan. 1977, pp. 238–252.

[79] J. S. Collofello and L. Cousins, "Towards automatic software fault localization through decision-to-decision path analysis," in *Proc. Nat. Comput. Conf.*, Jun. 1987, pp. 539–544.

[80] D. S. Coutant, S. Meloy, and M. Ruscetta, "DOC: A practical approach to source-level debugging of globally optimized code," in *Proc. ACM SIGPLAN Conf. Program Language Des. Implementation*, Atlanta, GA, USA, Jul. 1988, pp. 125–134.

[81] C. Csallner and Y. Smaragdakis, "Check 'n' crash: Combining static checking and testing," in *Proc. IEEE Int. Conf. Softw. Eng.*, May 2005, pp. 422–431.

[82] J. Cunha, M. Erwig, and J. Saraiva, "Automatically inferring classsheet models from spreadsheets," in *Proc. IEEE Symp. Visual Language Human-Centric Comput.*, Leganes-Madrid, Spain, Sep. 2010, pp. 93–100.

[83] J. Cunha, J. P. Fernandes, J. Mendes, and J. Saraiva, "MDSheet: A framework for model-driven spreadsheet engineering," in *Proc. IEEE Int. Conf. Softw. Eng.*, Zurich, Switzerland, Jun. 2012, pp. 1412–1415.

[84] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for Java," in *Proc. Eur. Conf. Object-Oriented Program.*, Glasgow, U.K., Jul. 2005, pp. 528–550.

[85] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proc. Int. Conf. Autom. Softw. Eng.*, Atlanta, GA, USA, Nov. 2007, pp. 433–436.

[86] (2015, Jun. 10). Delta tool [Online]. Available: http://delta.tigris.org/

[87] B. C. Dean, W. B. Pressly, B. A. Malloy, and A. A. Whitley, "A linear programming approach for automated localization of multiple faults," in *Proc. Int. Conf. Autom. Softw. Eng.*, Auckland, New Zealand, Nov. 2009, pp. 640–644.

[88] H. A. de Souza and M. L. Chaim, "Adding context to fault localization with integration coverage," in *Proc. Int. Conf. Autom. Softw. Eng.*, Silicon Valley, CA, USA, Nov. 2013, pp. 628–633.

[89] V. Debroy, "Towards the automation of program debugging," Ph.D. dissertation, The Univ. Texas at Dallas, Richardson, TX, USA, 2011.

[90] V. Debroy and W. E. Wong, "Insights on fault interference for programs with multiple bugs," in *Proc. Int. Symp. Softw. Rel. Eng.*, Karnataka, India, Nov. 2009, pp. 165–174.

[91] V. Debroy and W. E. Wong, "A consensus-based strategy to improve the quality of fault localization," *Softw.: Practice Experience*, vol. 43, no. 8, pp. 989–1011, Aug. 2013.

[92] V. Debroy and W. E. Wong, "On the consensus-based application of fault localization techniques," in *Proc. IEEE Annu. Comput. Softw. Appl. Conf. Workshops*, Munich, Germany, Jul. 2011, pp. 506–511.

[93] V. Debroy and W. E. Wong, "On the equivalence of certain fault localization techniques," in *Proc. ACM Symp. Appl. Comput.*, TaiChung, Taiwan, Mar. 2011, pp. 1457–1463.

[94] V. Debroy, W. E. Wong, X. Xu, and B. Choi, "A grouping-based strategy to improve the effectiveness of fault localization techniques," in *Proc. Int. Conf. Quality Softw.*, Zhangjiajie, China, Jul. 2010, pp. 13–22.

[95] V. Debroy and W. E. Wong, "Combining mutation and fault localization for automated program debugging," *J. Syst. Softw.*, vol. 90, pp. 45–60, Apr. 2014.

[96] R. A. DeMillo, H. Pan, and E. H. Spafford, "Critical slicing for software fault localization," in *Proc. Int. Symp. Softw. Testing Anal.*, San Diego, CA, USA, Jan. 1996, pp. 121–134.

[97] R. A. DeMillo, H. Pan, and E. H. Spafford, "Failure and fault analysis for software debugging," in *Proc. IEEE Annu. Comput. Softw. Appl. Conf.*, 1977, pp. 515–521.

[98] J. D. DeMott, "Enhancing automated fault discovery and analysis," Ph.D. dissertation, Michigan State Univ., East Lansing, MI, USA, 2012.

[99] T. Denmat, M. Ducassé, and O. Ridoux, "Data mining and cross-checking of execution traces," in *Proc. Int. Conf. Autom. Softw. Eng.*, Long Beach, CA, USA, Nov. 2005, pp. 396–399.

[100] E. Diaz, J. Tuya, and R. Blanco, "Automated software testing using a meta-heuristic technique based on Tabu search," in *Proc. Conf. Autom. Softw. Eng.*, Montreal, QC, Canada, Oct. 2003, pp. 310–313.

[101] W. Dickinson, D. Leon, and A. Fodgurski, "Finding failures by cluster analysis of execution profiles," in *Proc. Int. Conf. Auto. Softw. Eng.*, Toronto, ON, Canada, May 2001, pp. 339–348.

[102] N. DiGiuseppe and J. A. Jones "On the influence of multiple faults on coverage-based fault localization," in *Proc. Int. Symp. Softw. Testing Anal.*, Toronto, ON, Canada, Jul. 2011, pp. 210–220.

[103] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Trans. Softw. Eng.*, vol. 32, no. 9, pp. 733–752, Sep. 2006.

[104] (2015, Mar. 24). DrDebug [Online]. Available: www.drdebug.org

[105] J. C. Edwards, "Method, system, and program for logging statements to monitor execution of a program," U.S. Patent 6 539 501, Mar. 2003.

[106] D. Engler and K. Ashcraft, "RacerX: Effective, static detection of race conditions and deadlocks," in *Proc. ACM Symp. Operating Syst. Principles*, Bolton Landing, NY, USA, Oct. 2003, pp. 237–252.

[107] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, pp. 99–123, Feb. 2001.

[108] (2015, Oct. 6). Eclipse Test & Performance Tools Platform Project [Online]. Available: http://www.eclipse.org/tptp

[109] E. Ermis, M. Schaf, and T. Wies, "Error invariants," in *Proc. Int. Symp. Formal Methods*, Paris, France, Aug. 2012, pp. 187–201.

[110] A. Farzan and P. Madhusudan. "Causal atomicity," in *Proc. Int. Conf. Comput. Aided Verification*, Seattle, WA, USA, Aug. 2006, pp. 315–328.

[111] A. Farzan, P. Madhusudan, and F. Sorrentino. "Meta-analysis for atomicity violations under nested locking," in *Proc. Int. Conf. Comput. Aided Verification*, Grenoble, France, Jul. 2009, pp. 248–262.

[112] L. Fausett, *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1994.

[113] C. Flanagan and S. N. Freund, "Atomizer: A dynamic atomicity checker for multithreaded programs," in *Proc. ACM SIGPLAN-SIGACT Symp. Principles Programm. Languages*, Venice, Italy, Jan. 2004, pp. 256–267.

[114] C. Flanagan and S. N. Freund, "Adversarial memory for detecting destructive races," in Proc. ACM SIGPLAN Conf. Programm. Language Design Implementation, Toronto, ON, Canada, Jun. 2010, pp. 244–254.

[115] C. Flanagan, S. N. Freund, and S. Qadeer, "Exploiting purity for atomicity," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 275–291, May 2005.

[116] C Flanagan, S. N. Freund, and J. Yi, "Velodrome: A Sound and complete dynamic atomicity checker for multithreaded programs," in *Proc. ACM SIGPLAN Conf. Program. Language Design Implementation*, Tucson, AZ, USA, Jun. 2008, pp. 293–303.

[117] G. Friedrich, M. Stumptner, and F. Wotawa, "Model-based diagnosis of hardware designs," *Artif. Intell.*, vol. 111, nos. 1/2, pp. 3–39, Jul. 1999.

[118] Z. P. Fry, "Fault localization using textual similarities," Master thesis, Univ. Virginia, Charlottesville, VA, USA, 2011.

[119] Y. Gao, Z. Zhang, L. Zhang, C. Gong, and Z. Zheng, "A theoretical study: The impact of cloning failed test cases on the effectiveness of fault localization," in *Proc. Int. Conf. Quality Softw.*, Nanjing, China, Jul. 2013, pp. 288–291.

[120] (2015, Feb. 14). GNU gprof [Online]. Available: http://sourceware.org/binutils/docs/gprof/

[121] (2015, Jun. 6). GNU GDB Online]. Available: http://www.gnu.org/software/gdb/

[122] L. Gong, D. Lo, L. Jiang, and H. Zhang, "Diversity maximization speedup for fault," in *Proc. Int. Conf. Autom. Softw. Eng.*, Essen, Germany, Sep. 2012, pp. 30–39.

[123] L. Gong, D. Lo, L. Jiang, and H. Zhang, "Interactive fault localization leveraging simple user feedback," in *Proc. Int. Conf. Softw. Maintenance*, Trento, Italy, Sep. 2012, pp. 67–76.

[124] C. Gong, Z. Zheng, W. Li, and P. Hao, "Effects of class imbalance in test suites: An empirical study of spectrum-based fault localization," in *Proc. IEEE Annu. Comput. Softw. Appl. Conf. Workshops*, Izmir, Turkey, July 2012, pp. 470–475.

[125] C. Gong, Z. Zheng, Y. Zhang, Z. Zhang, and Y. Xue, "Factorising the multiple fault localization problem: Adapting single-fault localizer to multi-fault programs," in *Proc. Asia-Pacific Softw. Eng. Conf.*, Hong Kong, Dec. 2012, pp. 729–732.

[126] A. Gonzalez-Sanchez, E. Piel, H. Gross, and A. J. van Gemund, "Prioritizing tests for software fault localization," in *Proc. Int. Conf. Quality Softw.*, Zhangjiajie, China, Jul. 2010, pp. 42–51.

[127] A. Gonzalez-Sanchez, R. Abreu, H. Gross, and A. J. C. van Gemund, "An empirical study on the usage of testability information to fault localization in software," in *Proc. ACM Symp. Appl. Comput.*, TaiChung, Taiwan, Mar. 2011, pp. 1398–1403.

[128] A. Gonzalez-Sanchez, R. Abreu, H. Gross, and A. J. C. van Gemund, "Prioritizing tests for fault localization through ambiguity group reduction," in *Proc. Int. Conf. Autom. Softw. Eng.*, Lawrence, KS, USA, Nov. 2011, pp. 83–92.

[129] D. Gopinath, R. N. Zaeem, and S. Khurshid, "Improving the effectiveness of spectra-based fault localization using specifications," in *Proc. Int. Conf. Autom. Softw. Eng.*, Essen, Germany, Sep. 2012, pp. 40–49.

[130] C. Gouveia, J. Campos, and R. Abreu, "Using HTML5 visualizations in software fault localization," in *Proc. 1st IEEE Working Conf. Softw. Vis.*, Eindhoven, The Netherlands, Sep. 2013, pp. 1–10.

[131] W. B. Gregory, "Logic programs for consistency-based diagnosis," Ph.D. dissertation, Carleton Univ., Ottawa, ON, Canada, 1994.

[132] A. Griesmayer, "Debugging software: From verification to repair," Ph.D. dissertation, Graz Univ. Technol., Graz, Austria, 2007.

[133] A. Griesmayer, S. Staber, and R. Bloem, "Automated fault localization for C programs," *Electron. Notes Theoretical Comput. Sci.*, vol. 174, no. 4, pp. 95–111, May 2007.

[134] A. Griesmayer, S. Staber, and R. Bloem, "Fault localization using a model checker," *Softw. Testing, Verification Rel.*, vol. 20, no. 2, pp. 149–173, Jun. 2010.

[135] A. Groce, "Error explanation and fault localization with distance metrics," Ph.D. dissertation, Carnegie Mellon Univ., Pittsburgh, PA, USA, 2005.

[136] A. Groce and W. Visser, "What went wrong: Explaining counterexamples," in *Proc. Int. Conf. Model Checking Softw.*, Portland, USA, May 2003, pp. 121–136.

[137] A. Groce, S. Chaki, D. Kroening, and O. Strichman, "Error explanation with distance metrics," *Int. J. Softw. Tools Technol. Transfer*, vol. 8, no. 3, pp. 229–247, Jun. 2006.

[138] A. Groce, D. Kroening, and F. Lerda, "Understanding counterexample with explain," *Comput. Aided Verification*, vol. 3114, pp. 453–456, Jul. 2004.

[139] M. Grottke and K. S. Trivedi, "A classification of software faults," in *Proc. 16th Int. Symp. Softw. Rel. Eng.*, Chicago, IL, USA, Nov. 2005, pp. 419–420.

[140] L. Guo, A. Roychoudhury, and T. Wang, "Accurately choosing execution runs for software fault localization," in *Proc. Int. Conf. Compiler Construction*, Vienna, Austria, Mar. 2006, pp. 80–95.

[141] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating faulty code using failure-inducing chops," in *Proc. Int. Conf. Autom. Softw. Eng.*, Long Beach, CA, USA, Nov. 2005, pp. 263–272.

[142] T. Gyimothy, A. Beszedes, and I. Forgacs, "An efficient relevant slicing method for debugging," in *Proc. Eur. Softw. Eng. Conf., Held Jointly with ACM SIGSOFT Symp. Found. Softw. Eng.*, Toulouse, France, Sep. 1999, pp. 303–321.

[143] M. Hamill and K. Goseva-Popstojanova, "Common trends in software fault and failure data," *IEEE Trans. Softw. Eng.*, vol. 35, no. 4, pp. 484–496, Jul./Aug. 2009.

[144] C. Hammer, J. Dolby, M. Vaziri, and F. Tip, "Dynamic detection of atomic-set-serializability violations," in *Proc. Int. Conf. Softw. Eng.*, Leipzig, Germany, May 2008, pp. 231–240.

[145] D. Hao, L. Zhang, L. Zhang, J. Sun, and H. Mei, "VIDA: Visual interactive debugging," in *Proc. Int. Conf. Softw. Eng.*, Vancouver, BC, Canada, May 2009, pp. 583–586.

[146] D. Hao, L. Zhang, H. Zhong, H. Mei, and J. Sun, "Eliminating harmful redundancy for testing-based fault localization using test suite reduction: An experimental study," in *Proc. Int. Conf. Softw. Maintenance*, Budapest, Hungary, Sep. 2005, pp. 683–686.

[147] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo, "Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem," in *Proc. Int. Conf. Softw. Testing, Verification, Validation Workshops*, Paris, France, Apr. 2010, pp. 182–191.

[148] P. McMinn, M. Stevenson, and M. Harman, "Reducing qualitative human oracle costs associated with automatically generated test data," in *Proc. Int. Workshop Softw. Test Output Validation*, Trento, Italy, Jul. 2010, pp. 1–4.

[149] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An empirical investigation of the relationship between spectra differences and regression faults," *J. Softw. Testing, Verification Rel.*, vol. 10, no. 3, pp. 171–194, Sep. 2000.

[150] M. Hauswirth and T. M. Cillmbi, "Low-overhead memory leak detection using adaptive statistical profiling," in *Proc. Int. Conf. Archit. Support Programm. Lang. Oper. Syst.*, Boston, MA, USA, Oct. 2004, pp. 156–164.

[151] M. Hays, "A fault-based model of fault localization techniques," Ph.D. dissertation, Univ. Kentucky, Lexington, KY, USA, 2014.

[152] H. He and N. Gupta, "Automated debugging using path-based weakest preconditions," in *Proc. 7th Int. Conf. Fundam. Approaches Softw. Eng.*, Barcelona, Spain, Mar. 2004, pp. 267–280.

[153] H. He, "Automated debugging using path-based weakest preconditions," Master thesis, Univ. Arizona, Tucson, AZ, USA, 2004

[154] R. Hecht-Nielsen, "Theory of the back-propagation neural network," in *Proc. Int. Joint Conf. Neural Netw.*, Washington, DC, USA, Jun 1989, pp. 593–605.

[155] J. Hennessy, "Symbolic debugging of optimized code," *ACM Trans. Program. Language Syst.*, vol. 4, no. 3, pp. 323–344, Jul. 1982.

[156] F. Hermans, M. Pinzger, and A. van Deursen, "Supporting professional spreadsheet users by generating leveled dataflow diagrams," in *Proc. Int. Conf. Softw. Eng.*, Waikiki, HI, USA, May 2011, pp. 451–460.

[157] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting and visualizing inter-worksheet smells in spreadsheets," in *Proc. Int. Conf. Softw. Eng.*, Zurich, Switzerland, Jun. 2012, pp. 441–451.

[158] F. Hermans, B. Sedee, M. Pinzger, and A. van Deursen, "Data clone detection and visualization in spreadsheets," in *Proc. Int. Conf. Softw. Eng.*, San Francisco, CA, USA, May 2013, pp. 292–301.

[159] R. M. Hierons, "Avoiding coincidental correctness in boundary value analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 3, pp. 227–241, Jul. 2006.

[160] R. M. Hierons, "Oracles for distributed testing," *IEEE Trans. Softw. Eng.*, vol. 38, no. 3, pp. 629–641, May/Jun. 2012.

[161] R. M. Hierons, "Verdict functions in testing with a fault domain or test hypotheses," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 4, article 14, Jul. 2009.

[162] K. Hodnigg, and R. T. Mittermeir, "Metrics-based spreadsheet visualization: Support for focused maintenance," in *Proc. EuSpRIG Conf. Pursuit of Spreadsheet Excellence*, 2008, pp. 79–94.

[163] B. Hofer, "From fault localization of programs written in 3rd level language to spreadsheets," Ph.D. dissertation, Graz Univ. Technol., Graz, Austria, 2013.

[164] B. Hofer, A. Riboira, F. Wotawa, R. Abreu, and E. Getzner, "On the empirical evaluation of fault localization techniques for spreadsheets," *Fundam. Approaches Softw. Eng.*, vol. 7793, pp. 68–82, Mar. 2013.

[165] B. Hofer and F. Wotawa, "Spectrum enhanced dynamic slicing for better fault localization," in *Proc. Eur. Conf. Artif. Intell.*, Montpellier, France, Aug. 2012, , pp. 420–425.

[166] D. Hovemeyer, "Simple and effective static analysis to find bugs," Ph.D. dissertation, Univ. Maryland, College Park, MD, USA, 2005.

[167] P. Hu, "Automated fault localization: A statistical predicate analysis approach," Ph.D. dissertation, the Univ. Hong Kong, Hong Kong, 2006.

[168] P. Hu, Z. Zhang, W. K. Chan, and T. H. Tse, "Fault localization with non-parametric program behavior model," in *Proc. Int. Conf. Quality Softw.*, Oxford, UK, Aug. 2008, pp. 385–395.

[169] D. Jannach and U. Engler, "Toward model-based debugging of spreadsheet programs," in *Proc. Joint Conf. Knowl.-Based Softw. Eng.*, Kaunas, Lithuania, Aug. 2010, pp. 252–264.

[170] T. Janssen, R. Abreu, and A. J.C. van Germund, "Zoltar: A spectrum-based fault localization tool," in *Proc. ESEC/FSE Workshop Softw. Integr. Eval.*, Amsterdam, The Netherlands, Aug. 2009, pp. 23–30.

[171] D. Jeffrey, "Dynamic state alteration techniques for automatically locating software errors," Ph.D. dissertation, Univ. California Riverside, Riverside, CA, USA, 2009.

[172] D. Jeffrey, N. Gupta and R. Gupta, "Fault localization using value replacement," in *Proc. Int. Symp. Softw. Testing Anal.*, Seattle, WA, USA, Jul. 2008, pp. 167–178.

[173] D. Jeffrey, N. Gupta and R. Gupta, "Effective and efficient localization of multiple faults using value replacement," in *Proc. Int. Conf. Softw. Maintenance*, Edmonton, AB, Canada, Sep. 2009, pp. 221–230.

[174] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta, "BugFix: A learning-based tool to assist developers in fixing bugs," in *Proc. Int. Conf. Program Comprehension*, Vancouver, BC, Canada, May 2009, pp. 70–79.

[175] B. Jiang and W. K. Chan, "On the integration of test adequacy, test case prioritization, and statistical fault localization," in *Proc. Int. Conf. Quality Softw.*, Zhangjiajie, China, Jul. 2010, pp. 377–384.

[176] B. Jiang, W. K. Chan, and T. H. Tse, "On practical adequate test suites for integrated test case prioritization and fault localization," in *Proc. Int. Conf. Quality Softw.*, Madrid, Spain, Jul. 2011, pp. 21–30.

[177] B. Jiang, K. Zhai, W. K. Chan, T. H. Tse, and Z. Zhang, "On the adoption of MC/DC and control-flow adequacy for a tight integration of program testing and statistical fault localization," *Inf. Softw. Technol.*, vol. 55, no. 5, pp. 897–917, May 2013.

[178] B. Jiang, Z. Zhang, W. K. Chan, T. H. Tse, and T. Y. Chen, "How well does test case prioritization integrate with statistical fault localization?" *J. Inf. Softw. Technol.*, vol. 54, no. 7, pp. 739–758, Jul. 2012.

[179] W. Jin and A. Orso, "F3: Fault localization for field failures," in *Proc. Int. Symp. Softw. Testing Anal.*, Lugano, Switzerland, Jul. 2013, pp. 213–223.

[180] M. Jose and R. Majumdar, "Bug-assist: Assisting fault localization in ANSI-C programs," in *Proc. Int. Conf. Comput. Aided Verification*, Snowbird, UT, USA, Jul. 2011, pp. 504–509.

[181] M. Jose and R. Majumdar, "Cause clue clauses: Error localization using maximum satisfiability," in *Proc. ACM SIGPLAN Conf. Programm. Language Des. Implementation*, Portland, OR, USA, Jun. 2011, pp. 437–446.

[182] H. Lee, "Spectral debugging," Ph.D. dissertation, The Univ. Melbourne, Melbourne, Vic., Australia, 2011.

[183] J. A. Jones, "Semi-automatic fault localization," Ph.D. dissertation, Georgia Inst. Technol., Atlanta, GA, USA, 2008.

[184] J. A. Jones, J. Bowring, and M. J. Harrold, "Debugging in parallel," in *Proc. ACM/SIGSOFT Int. Symp. Softw. Testing Anal.*, London, U.K., Jul. 2007, pp. 16–26.

[185] J. A. Jones, and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proc. Int. Conf. Autom. Softw. Eng.*, Long Beach, CA, USA, Nov. 2005, pp. 273–282.

[186] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization for fault localization," in *Proc. Workshop Softw. Vis., 23rd Int. Conf. Softw. Eng.*, Ontario, BC, Canada, May 2001, pp. 71–75.

[187] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proc. Int. Conf. Softw. Eng.*, Orlando, FL, USA, May 2002, pp. 467–477.

[188] X. Ju, S. Jiang, X. Chen, X. Wang, Y. Zhang, and H. Cao, "HSFal: Effective fault localization using hybrid spectrum of full slices and execution slices," *J. Syst. Softw.*, vol. 90, pp. 3–17, Apr. 2014.

[189] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, "MintHint: Automated synthesis of repair hints," in *Proc. Int. Conf. Softw. Eng.*, Hyderabad, India, May 2014, pp. 266–276.

[190] N. Kidd, T. Reps, J. Dolby, and M. Vaziri, "Finding concurrency-related bugs using random isolation," in *Proc. Int. Conf. Verification, Model Checking, Abstract Interpretation*, Savannah, GA, USA, Jan. 2009, pp. 198–213.

[191] J. Kim and E. Lee, "Empirical evaluation of existing algorithms of spectrum based fault localization," in *Proc. Int. Conf. Inf. Netw.*, Feb. 2014, pp. 346–351.

[192] Á. Kiss, J. Jász, and T. Gyimóthy, "Using dynamic information in the interprocedural static slicing of binary executables," *Softw. Quality Control*, vol. 13, no. 3, pp. 227–245, Sep. 2005.

[193] J. de Kleer, "Diagnosing multiple persistent and intermittent faults," in *Proc. Int. Joint Conf. Artif. Intell.*, Pasadena, USA, Jul. 2009, pp. 733–738.

[194] J. de Kleer and B.C. Williams, "Diagnosing multiple faults," *Artif. Intell.*, vol. 32, no. 1, pp. 97–130, Apr. 1987.

[195] A. J. Ko and B. A. Myers, "Debugging reinvented: Asking and answering why and why not questions about program behavior," in *Proc. Int. Conf. Softw. Eng.*, Leipzig, Germany, May 2008, pp. 301–310.

[196] D. Kob and F. Wotawa, "Introducing alias information into model-based debugging," in *Proc. Eur. Conf. Artif. Intell.*, Valencia, Spain, Aug. 2004, pp. 833–837.

[197] D. Kob, "Extended modeling for automated fault localization in object-oriented software," Ph.D. Dissertation, Graz Univ. Technol., Graz, Austria, 2005.

[198] F. Koca, H. Sozer, and R. Abreu, "Spectrum-based fault localization for diagnosing concurrency faults," in *Proc. Int. Conf. Testing Softw. Syst.*, Istanbul, Turkey, Nov. 2013, pp. 239–254.

[199] R. R. Kompella, "Fault localization in backbone networks," Ph.D. dissertation, Univ. California, CA, USA, 2007.

[200] R. KÖnighofer and R. Bloem, "Automated error localization and correction for imperative programs," in *Proc. Int. Conf. Formal Methods Comput.-Aided Des.*, Austin, TX, USA, Oct. 2011, pp. 91–100.

[201] B. Korel, "PELAS – program error-locating assistant system," *IEEE Trans. Softw. Eng.*, vol. 14, no. 9, pp. 1253–1260, Sep. 1988.

[202] B. Korel and J. Laski, "Dynamic program slicing," *Inf. Process. Lett.*, vol. 29, no. 3, pp. 155–163, Oct. 1988.

[203] B. Korel and J. Laski, "STAD: A system for testing and debugging: User perspective," in *Proc. 2nd Workshop Softw. Testing, Verification, Anal.*, Jul. 1988, pp. 13–20.

[204] B. Korel and S. Yalamanchili, "Forward computation of dynamic program slices," in *Proc. Int. Symp. Softw. Testing Anal.*, Seattle, WA, USA, Aug. 1994, pp. 66–79.

[205] J. Krinke, "Slicing, chopping, and path conditions with barriers," *Softw. Quality Control*, vol. 12, no. 4, pp. 339–360, Dec. 2004.

[206] C. Kuhnert, "Data-driven methods for fault localization in process technology," Ph.D. dissertation, Karlsruhe Inst. Technol., Karlsruhe, Germany, 2013.

[207] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue, "Experimental evaluation of program slicing for fault localization," *Empirical Softw. Eng.*, vol. 7, no. 1, pp. 49–76, Mar. 2002.

[208] N. Lazaar, "CPTEST: A framework for the automatic fault detection, localization and correction of constraint programs," in *Proc. 4th Int. Conf. Softw. Testing, Verification, Validation*, Berlin, Germany, Mar. 2011, pp. 320–321.

[209] T. B. Le and D. Lo, "Will fault localization work for these failures? an automated approach to predict effectiveness of fault localization tools," in *Proc. 29th IEEE Int. Conf. Softw. Maintenance*, Eindhoven, The Netherland, Sep. 2013, pp. 310–319.

[210] T. B. Le, F. Thung, and D. Lo, "Theory and practice, do they match? a case with spectrum-based fault localization," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Eindhoven, The Netherland, Sep. 2013, pp. 380–383.

[211] C. C. Lee, P. C. Chung, J. R. Tsai, and C. I. Chang, "Robust radial basis function neural networks," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 29, no. 6, pp. 674–685, Dec. 1999.

[212] H. J. Lee, L. Naish, and K. Ramamohanarao, "Study of the relationship of bug consistency with respect to performance of spectra metrics," in *Proc. Int. Conf. Comput. Sci. Inf. Technol.*, Aug. 2009, pp. 501–508.

[213] H. J. Lee, L. Naish, and K. Ramamohanarao, "Effective software bug localization using spectral frequency weighting function," in *Proc. Annu. IEEE Int. Comput. Softw. Appl. Conf.*, Seoul, Korea, Jul. 2010, pp. 218–227.

[214] Y. Lei, X. Mao, Z. Dai, and C. Wang, "Effective statistical fault localization using program slices," in *Proc. Annu. IEEE Int. Comput. Softw. Appl. Conf.*, Izmir, Turkey, Jul. 2012, pp. 1–10.

[215] D. Lewis, "Causation," *J. Philosophy*, vol. 70, no. 17, pp. 556–567, Oct. 1973.

[216] D. Lewis, *Counterfactuals*. Cambridge, MA, USA: Harvard Univ. Press, 1973.

[217] F. Li, W. Huo, C. Chen, L. Zhong, X. Feng, and Z. Li, "Effective fault localization based on minimum debugging frontier set," in *Proc. Int. Symp. Code Gener. Optimization*, Shenzhen, China, Feb. 2013, pp. 23–27.

[218] Y. Li and C. Liu, "Using cluster analysis to identify coincidental correctness in fault localization," in *Proc. Int. Conf. Comput. Inf. Sci.*, Nanchong, China, May 2012, pp. 357–360.

[219] L. Lian, S. Kusumoto, T. Kikuno, K. Matsumofo, and K. Torii, "A new fault localizing method for the program debugging process," *Inf. Softw. Technol.*, vol. 39, no. 4, pp. 271–284, Apr. 1997.

[220] D. Liang and M. J. Harrold, "Equivalence analysis and its application in improving the efficiency of program slicing," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 3, pp. 347–383, Jul. 2002.

[221] B. Liblit, "Cooperative bug isolation," Ph.D. dissertation, Univ. California at Berkeley, Berkeley, CA, USA, 2004.

[222] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proc. ACM SIGPLAN Conf. Programm. Language Design Implementation*, Chicago, IL, USA, Jul. 2005, pp. 15–26.

[223] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 10, pp. 831–848, Oct. 2006.

[224] C. Liu and J. Han, "Failure proximity: A fault localization-based approach," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Portland, OR, USA, Nov. 2006, pp. 286–295.

[225] C. Liu, X. Yan, H. Yu, J. Han, and P. Yu, "Mining behavior graphs for 'Backtrace' of noncrashing bugs," in *Proc. SIAM Int. Conf. Data Mining*, Philadelphia, PA, USA, Apr. 2014, pp. 286–297.

[226] C. Liu, X. Zhang, and J. Han, "A systematic study of failure proximity," *IEEE Trans. Softw. Eng.*, vol. 34, no. 6, pp. 826–843, Nov./Dec. 2008.

[227] C. Liu, X. Zhang, J. Han, Y. Zhang, and B. K. Bhargava, "Indexing noncrashing failures: A dynamic program slicing-based approach" in *Proc. Int. Conf. Softw. Maintenance*, Paris, France, Oct. 2007, pp. 455–464.

[228] Y. Liu, "Automated analysis of energy efficiency and performance for mobile application," Ph.D. dissertation, the Hong Kong Univ. Sci. Technol., Hong Kong, 2015.

[229] S. Lu, "Understanding, detecting, and exposing concurrency bugs," Ph.D. dissertation, Univ. Illinois at Urbana-Champaign, Champaign, IL, USA, 2008.

[230] D. Lucia Lo, L. Jiang, F. Thung, and A. Budi, "Extended comprehensive study of association measures for fault localization," *J. Softw.: Eval. Process*, vol. 26, no. 2, pp. 172–219, Feb. 2014.

[231] F. Lucia Thung, D. Lo, and L. Jiang, "Are faults localizable?" in *Proc. IEEE Working Conf. Mining Softw. Repositories*, Zurich, Switzerland, Jun. 2012, pp. 74–77.

[232] D. Lucia Lo, and X. Xia, "Fusing fault localizers," in *Proc. IEEE Int. Conf. Autom. Softw. Eng.*, Vasteras, Sweden, Sep. 2014, pp. 127–138.

[233] L. Lucia, "Ranking-based approaches for localizing faults," Ph.D. dissertation, Singapore Manage. Univ., Singapore, 2014.

[234] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Source code retrieval for bug localization using Latent Dirichlet Allocation," in *Proc. Working Conf. Reverse Eng.*, Antwerp, Belgium, Oct. 2008, pp. 155–164.

[235] J. R. Lyle and M. Weiser, "Automatic program bug location by program slicing," in *Proc. Int. Conf. Comput. Appl.*, 1987, pp. 877–883.

[236] P. Machado, J. Campos, and R. Abreu, "MZoltar: Automatic debugging of android applications," in *Proc. Int. Workshop Softw. Develop. Lifecycle Mobile*, Saint Petersburg, Russia, Aug. 2013, pp. 9–16.

[237] X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang, "Slice-based statistical fault localization," *J. Syst. Softw.*, vol. 89, pp. 51–62, Mar. 2014.

[238] K. Maruyama and M. Terada, "Debugging with reverse Watchpoint," in *Proc. Int. Conf. Quality Softw.*, Dallas, TX, USA, Nov. 2003, p. 116.

[239] W. Masri and R. A. Assi "Cleansing test suites from coincidental correctness to enhance fault-localization," in *Proc. Int. Conf. Softw. Testing, Verification, Validation*, Paris, France, Apr. 2010, pp. 369–399.

[240] W. Masri and R. A. Assi, "Prevalence of coincidental correctness and mitigation of its impact on fault localization," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 1, article 8, Feb. 2014.

[241] C. Mateis, M. Stumptner, D. Wieland, and F. Wotawa, "JADE - AI support for debugging Java programs," in *Proc. IEEE Int. Conf. Tools Artif. Intell.*, Washington, DC, USA, Nov. 2002, pp. 62–69.

[242] C. Mateis, M. Stumptner, and F. Wotawa, "Modeling Java programs for diagnosis," in *Proc. Eur. Conf. Artif. Intell.*, Berlin, Germany, 2000, pp. 171–175.

[243] W. Mayer, R. Abreu, M. Stumptner, and A. J. C. Van Gemund, "Prioritizing model-based debugging diagnostic reports," in *Proc. Int. Workshop Principles Diagnosis*, Blue Mountains, N.S.W., Australia, Sep. 2008, pp. 127–134.

[244] W. Mayer and M. Stumptner, "Modeling programs with unstructured control flow for debugging," in *Proc. Conf. Can. Soc. Comput. Stud. Intell.*, Calgary, AB, Canada, May 2002, pp. 107–118.

[245] W. Mayer and M. Stumptner, "Debugging program exceptions," in *Proc. Int. Workshop Principles Diagnosis*, 2003, pp. 119–124.

[246] W. Mayer and M. Stumptner, "Approximate modeling for debugging of program loops," in *Proc. Int. Workshop Principles Diagnosis*, Carcassonne, France, Jun. 2004, pp. 87–92.

[247] W. Mayer and M. Stumptner, "Abstract interpretation of programs for model-based debugging," in *Proc. Int. Joint Conf. Artif. Intell.*, Hyderabad, India, Jan. 2007, pp. 471–476.

[248] W. Mayer and M. Stumptner, "Model-based debugging using multiple abstract models," in *Proc. Int. Workshop Automated Debugging*, Monterey, CA, USA, Sep. 2003, pp. 55–70.

[249] W. Mayer and M. Stumptner, "Model-based debugging: State of the art and future challenges," *Electron. Notes Theoretical Comput. Sci.*, vol. 174, no. 4, pp. 61–82, May 2007.

[250] W. Mayer and M. Stumptner, "Evaluating models for model-based debugging," in *Proc. ACM Int. Conf. Autom. Softw. Eng.*, L'Aquila, Italy, Sep. 2008, pp. 128–137.

[251] W. Mayer, M. Stumptner, D. Wieland, and F. Wotawa, "Can AI help to improve debugging substantially? debugging experiences with value-based models," in *Proc. Eur. Conf. Artif. Intell.*, Lyon, France, Jul. 2002, pp. 417–421.

[252] W. Mayer, M. Stumptner, D. Wieland, and F. Wotawa, "Towards an integrated debugging environment," in *Proc. Eur. Conf. Artif. Intell.*, Lyon, France, Jul. 2002, pp. 422–426.

[253] S. McMaster and A. Memon, "Fault detection probability analysis for coverage-based test suite reduction," in *Proc. Int. Conf. Softw. Maintenance*, Paris, France, Oct. 2007, pp. 335–344.

[254] Y. Miao, Z. Chen, S. Li, Z. Zhao and Y. Zhou, "Identifying coincidental correctness for fault localization clustering test cases," presented at the *Int. Conf. Software Engineering Knowledge Engineering*, San Francisco, CA, USA, Jul. 2012

[255] (2015, Sep. 10). Microsoft Visual Studio Debugger [Online]. Available: https://msdn.microsoft.com/en-us/library/sc65sadd.aspx

[256] V. Modi, S. Roy, and S. K. Aggarwal, "Exploring program phases for statistical bug localization," in *Proc. ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng.*, Seattle, WA, USA, Jun. 2013, pp. 33–40.

[257] D. P. Mohapatra, R. Mall, and R. Kumar, "An edge marking technique for dynamic slicing of object-oriented programs," in *Proc. Int. Comput. Softw. Appl. Conf.*, Hong Kong, Sep. 2004, pp. 60–65.

[258] M. Monperrus, "A critical review of 'automated patch generation learned from human-written patches' essay on the problem statement and the evaluation of automatic software repair," in *Proc. Int. Conf. Softw. Eng.*, Hyderabad, India, Jun. 2014, pp. 234–242.

[259] S. Moon, "Effective software fault localization using dynamic program behaviors," Master thesis, Korea Adv. Inst. Sci. Technol., Daejeon, South Korea, 2014.

[260] J. C. Munson and T. M. Khoshgoftaar, "The detection of fault-prone programs," *IEEE Trans. Softw. Eng.*, vol. 18, no. 5, pp. 423–433, May 1992.

[261] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing Heisenbugs in concurrent programs," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, San Diego, CA, USA, Dec. 2008, pp. 267–280.

[262] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for Java," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, Ottawa, ON, Canada, Jun. 2006, pp. 308–319.

[263] L. Naish, H. J. Lee, and K. Ramamohanarao, "Spectral debugging with weights and incremental ranking," in *Proc. Asia-Pacific Softw. Eng. Conf.*, Batu Ferringhi, Malaysia, Dec. 2009, pp. 168–175.

[264] L. Naish, and H. J. Lee, "Duals in spectral fault localization," in *Proc. Australian Softw. Eng. Conf.*, Melbourne, Vic., Australia, Jun. 2013, pp. 51–59.

[265] L. Naish, H. J. Lee, and K. Ramamohanarao, "Spectral debugging: How much better can we do," in *Proc. Australian Softw. Eng. Conf.*, Melbourne, Vic., Australia, Jun. 2012, pp. 96–106.

[266] L. Naish, H. J. Lee, and K. Ramamohanarao, "Statements versus predicate in spectral bug localization," in *Proc. Asia-Pacific Softw. Eng. Conf.*, Sydney, N.S.W., Australia, Nov. 2010, pp. 375–384.

[267] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *J. ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, article 11, Aug. 2011.

[268] A. S. Namin, J. H. Andrews, and Y. Labiche, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Softw. Eng.*, vol. 32, no. 8, pp. 608–624, Aug. 2006.

[269] S. Nessa, M. Abedin, W. Eric Wong, L. Khan, and Y. Qi, "Fault localization using N-gram analysis," in *Proc. Int. Conf. Wireless Algorithms, Syst. Appl.*, Aug. 2009, pp. 548–559.

[270] H. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *Proc. Int. Conf. Softw. Eng.*, San Francisco, CA, USA, May 2013, pp. 772–781.

[271] M. Nica, "On the use of constraints in automated program debugging – from foundations to empirical results," Ph.D. dissertation, Graz Univ. Technol., Graz, Austria, 2010.

[272] M. Nica, S. Nica, and F. Wotawa, "On the use of mutations and testing for debugging," *Softw., Practice Experience*, vol. 43, no. 9, pp. 1121–1142, Sep. 2013.

[273] C. Nie and H. Leung, "The minimal failure-causing schema of combinatorial testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, article 15, Sep. 2011.

[274] X. Niu, C. Nie, Y. Lei, and A. T. Chan, "Identifying failure-inducing combinations using tuple relationship," in *Proc. Int. Conf. Softw. Testing, Verification, Validation Workshops*, Mar. 2013, pp. 271–280.

[275] G. J. Pai and J. B. Dugan, "Empirical analysis of software fault content and fault proneness using Bayesian methods," *IEEE Trans. Softw. Eng.*, vol. 33, no. 10, pp. 675–686, Oct. 2007.

[276] H. Pan, "Software debugging with dynamic instrumentation and test-based knowledge," Ph.D. dissertation, Purdue Univ., West Lafayette, IN, USA, 1993.

[277] H. Pan and E. Spafford, "Heuristics for automatic localization of software faults," Purdue Univ., West Lafayette, IN, USA, Tech. Rep. SERC-TR-116-P, 1992.

[278] R. Panko, "Facing the problem of spreadsheet errors," *Decision Line*, vol. 37, no. 5, pp. 8–10, 2006.

[279] R. R. Panko, "Applying code inspection to spreadsheet testing," *J. Manage. Inf. Syst.*, vol. 16, no. 2, pp. 159–176, Sep. 1999.

[280] R. R. Panko and N. Ordway, "Sarbanes-Oxley: What about all the spreadsheets?" *CoRR*, abs/0804.0797, 2008.

[281] R. R. Panko, "Spreadsheet errors: What we know. What we think we can do," in *Proc. EuSpRIG*, 2000, pp. 1–9.

[282] S. Park, S. Lu, and Y. Zhou, "CTrigger: Exposing atomicity violation bugs from their hiding places," in *Proc. Int. Conf. Architectural Support Program. Language*, Washington, D.C., USA, Mar. 2009, pp. 25–36.

[283] S. Park, "Debugging non-deadlock concurrency bugs," in *Proc. Int. Symp. Softw. Testing Anal.*, Lugano, Switzerland, Jul. 2013, pp. 358–361.

[284] S. Park, "Effective fault localization techniques for concurrent software," Ph.D. dissertation, Georgia Inst. Technol., Atlanta, GA, USA, 2014.

[285] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proc. Int. Symp. Softw. Testing Anal.*, Toronto, ON, Canada, Jul. 2011, pp. 199–209.

[286] F. Pastore, "Automatic diagnosis of software functional faults by means of inferred behavioral models," Ph.D. dissertation, Università degli Studi di Milano Bicocca, Milano, Italy, 2010.

[287] F. Pastore, L. Mariani, and A. Goffi, "RADAR: A tool for debugging regression problems in C/C++ software," in *Proc. Int. Conf. Softw. Eng.*, San Francisco, CA, USA, May 2013, pp. 1335–1338.

[288] J. Pearl, *Causality: Models, Reasoning, and Inference*. Cambridge , MA, USA: Cambridge Univ. Press, 2000.

[289] B. Peischl, "Automated source-level debugging of synthesizable VHDL designs," Ph.D. dissertation, Graz Univ. Technol., Graz, Austria, 2004.

[290] B. Peischl and F. Wotawa, "Automated source-level error localization in hardware designs," *IEEE Des. Test Comput.*, vol. 23, no. 1, pp. 8–19, Jan.–Mar. 2006.

[291] A. Perez, R. Abreu, and A. Riboira, "A dynamic code coverage approach to maximize fault localization efficiency," *J. Syst. Softw.*, vol. 90, pp. 18–28, Apr. 2014.

[292] A. Perez, "Dynamic code coverage with progressive detail levels," Ph.D. dissertation, Univ. Porto, Porto, Portugal, 2012.

[293] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Proc. Int. Conf. Softw. Eng.*, Portland, OR, USA, May 2003, pp. 465–475.

[294] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss, "Automated fault localization using potential invariants," in *Proc. Int. Workshop Autom. Debugging*, Monterey, USA, Sep. 2003, pp. 273–276.

[295] D. Qi, "Semantic analyses to detect and localize software regression errors," Ph.D. dissertation, Tsinghua Univ., Beijing, China, 2013.

[296] Y. Qi, X. Mao, Y. Lei and C. Wang, "Using automated program repair for evaluating the effectiveness of fault localization techniques," in *Proc. Int. Symp. Softw. Testing Anal.*, Lugano, Switzerland, Jul. 2013, pp. 191–201.

[297] J. Qian and B. Xu, "Scenario oriented program slicing," in *Proc. ACM Symp. Appl. Comput.*, Fortaleza, Brazil, Mar. 2008, pp. 748–752.

[298] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: A comparative study of generic and composite text models," in *Proc. Working Conf. Mining Softw. Repositories*, May 2011, pp. 43–52.

[299] P. Rao, Z. Zheng, T. Y. Chen, N. Wang, and K. Cai, "Impacts of test suite's class imbalance on spectrum-based fault localization techniques," in *Proc. 13th Int. Conf. Quality Softw.*, Nanjing, China, Jul. 2013, pp. 260–267.

[300] C. M. Reinhart and K. S. Rogoff, "Growth in a time of debt," *Amer. Econ. Rev.*, vol. 100, no. 2, pp. 573–578, Jan. 2010.

[301] R. Reiter, "A theory of diagnosis from first principles," *Artif. Intell.*, vol. 32, no. 1, pp. 57–95, Apr. 1987.

[302] E. Renieris, "A research framework for software-fault localization tools," Ph.D. dissertation, Brown Univ., Providence, RI, USA, 2005.

[303] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proc. Int. Conf. Autom. Softw. Eng.*, Montreal, QC, Canada, Oct. 2003, pp. 30–39.

[304] NIST Rep. (2002, May). The Economic Impacts of Inadequate Infrastructure for Software Testing [Online]. Available: http://www.abeacha.com/NIST_press_release_bugs_cost.htm

[305] 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 *Proc. ACM SIGSOFT Symp. Found. Softw. Eng.*, Zurich, Switzerland, Sep. 1997, pp. 432–449.

[306] N. Riaz, "Automated source-level debugging of synthesizable verilog designs," Graz University of Technology, Graz, Austria, Ph.D. dissertation, 2008.

[307] J. Rößler, G. Fraser, A. Zeller, and A. Orso, "Isolating failure causes through test case generation," in *Proc. Int. Symp. Softw. Testing Anal.*, Minneapolis, MN, USA, Jul. 2012, pp. 309–319.

[308] M. Rohr, "Workload-sensitive timing behavior analysis for fault localization in software systems," Ph.D. dissertation, Kiel Univ., Kiel, Germany, 2015.

[309] D. S. Rosenblum, "A practical approach to programming with assertions," *IEEE Trans. Softw. Eng.*, vol. 21, no. 1, pp. 19–31, Jan. 1995.

[310] D. S. Rosenblum, "Towards a method of programming with assertions," in *Proc. Int. Conf. Softw. Eng.*, Melbourne, Vic., Australia, 1992, pp. 92–104

[311] K. J. Rothermel, L. Li, C. DuParis, and M. Burnett, "What you see is what you test: A methodology for testing form-based visual programs," in *Proc. Int. Conf. Softw. Eng.*, Kyoto, Japan, Apr. 1998, pp. 198–207.

[312] S. Roychowdhry, "A mixed approach to spectrum-based fault localization using information theoretic foundations," Ph.D. dissertation, Univ. of Texas at Austin, Austin, TX, USA, 2013.

[313] C. Runciman and D. Wakeling, "Heap profiling of lazy functional programs," *J. Functional Program.*, vol. 3, no. 02, pp. 217–245, Apr. 1993.

[314] J. Ruthruff, E. Creswick, M. Burnett, C. Cook, S. Prabhakararao, M. Fisher, II, and M. Main, "End-user software visualizations for fault localization," in *Proc. ACM Symp. Softw. Vis.*, San Diego, CA, USA, Jun. 2003, pp. 123–132.

[315] R. K. Saha, M. Lease, S. Kunshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Proc. Int. Conf. Autom. Softw. Eng.*, Silicon Valley, CA, USA, Nov. 2013, pp. 345–355.

[316] S. K. Sahoo, "A novel invariants-based approach for automated software fault localization," Ph.D. dissertation, Univ. Illinois at Urbana-Champaign, Champaign, IL, USA, 2013.

[317] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve, “Using likely invariants for automated software fault localization,” in *Proc. Int. Conf. Architectural Support Program. Languages Operating Syst.*, Houston, TX, USA, Mar. 2013, pp. 139–152.

[318] A. Sanchez, “Cost optimizations in runtime testing and diagnosis,” Ph.D. dissertation, Delft Univ. Technol., Delft, The Netherlands, 2011.

[319] R. Santelices, “Change-effects analysis for effective testing and validation of evolving software,” Ph.D. dissertation, Georgia Inst. Technol., Atlanta, GA, USA, 2012.

[320] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, “Lightweight fault-localization using multiple coverage types,” in *Proc. Int. Conf. Softw. Eng.*, May 2009, pp. 56–66.

[321] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, “Eraser: A dynamic data race detector for multi-threaded programs,” *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.

[322] K. Sen, G. Rosu, and G. Agha, “Runtime safety analysis of multithreaded programs,” in *Proc. ACM SIGSOFT Symp. Found. Softw. Eng.*, Helsinki, Finland, Sep. 2003, pp. 337–346.

[323] F. Servant and J. A. Jones, “WhoseFault: Automatic developer-to-fault assignment through fault localization,” in *Proc. Int. Conf. Softw. Eng.*, Zurich, Switzerland, Jun. 2012, pp. 36–46.

[324] O. Shacham, M. Sagiv, and A. Schuster, “Scaling model checking of dataraces using dynamic information,” in *Proc. ACM SIGPLAN Symp. Principles Practice Parallel Program.*, Chicago, IL, USA, Jun. 2005, pp. 107–118.

[325] E. Shapiro, “Algorithmic program debugging,” Ph.D. dissertation, Yale Univ., New Haven, CT, USA, 1982.

[326] G. Shu, B. Sun, T. A. D. Henderson, and A. Podgurski, “JavaPDG: A new platform for program dependence analysis,” in *Proc. Int. Conf. Softw. Testing, Verification Validation*, Mar. 2013, pp. 408–415.

[327] G. Shu, “Statistical estimation of software reliability and failure-causing effect,” Ph.D. dissertation, Case Western Reserve Univ., Cleveland, OH, USA, 2014.

[328] J. Silva, “A survey on algorithmic debugging strategies,” *Adv. Eng. Softw.*, vol. 42, no. 11, pp. 976–991, Nov. 2011.

[329] F. Sorrentino, A. Farzan, and M. Parthasarathy, “Penelope: Weaving threads to expose atomicity violations,” in *Proc. ACM SIGSOFT Symp. Found. Softw. Eng.*, Santa Fe, NM, USA, Nov. 2010, pp. 37–46.

[330] M. Stridharan, S. J. Fink, and R. Bodik, “Thin slicing,” in *Proc. SIGPLAN Conf. Program. Language Des. Implementation*, San Diego, CA, USA, Jun. 2007, pp. 112–122.

[331] F. Steimann and M. Bertschler, “A simple coverage-based locator for multiple faults,” in *Proc. Int. Conf. Softw. Testing, Verification Validation*, Denver, CO, USA, Apr. 2009, pp. 366–375.

[332] F. Steimann and M. Frenkel, “Improving coverage-based localization of multiple faults using algorithms from integer linear programming,” in *Proc. Int. Symp. Softw. Rel. Eng.*, Dallas, TX, USA, Nov. 2012, pp. 121–130.

[333] F. Steimann, M. Frenkel, and R. Abreu, “Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators,” in *Proc. Int. Symp. Softw. Testing Anal.*, Lugano, Switzerland, Jul. 2013, pp. 314–324.

[334] C. D. Sterling and R. A. Olsson, “Automated bug isolation via program chipping,” in *Proc. Int. Symp. Autom. Anal.-Driven Debugging*, Monterey, CA, USA, Sep. 2005, pp. 23–32.

[335] M. Stumptner and F. Wotawa, “A survey of intelligent debugging,” *AI Commun.*, vol. 11, no. 1, pp. 35–51, Jan. 1998.

[336] M. Stumptner and F. Wotawa, “Debugging functional program,” in *Proc. Int. Joint Conf. Artif. Intell.*, Stockholm, Sweden, Aug. 1999, pp. 440–445.

[337] W. N. Sumner, “Automated failure explanation through execution comparison,” Ph.D. dissertation, Purdue Univ., West Lafayette, IN, USA, 2013.

[338] W. N. Sumner and X. Zhang, “Memory indexing: Canonicalizing addresses across executions,” in *Proc. Int. Symp. Found. Softw. Eng.*, Santa Fe, NM, USA, Nov. 2010, pp. 217–226.

[339] W. N. Sumner and X. Zhang, “Algorithms for automatically computing the causal paths of failures,” in *Proc. Int. Conf. Fundam. Approaches Softw. Eng.*, York, U.K., Mar. 2009, pp. 335–369.

[340] W. N. Sumner and X. Zhang, “Comparative causality: Explaining the differences between executions,” in *Proc. Int. Conf. Softw. Eng.*, San Francisco, CA, USA, May 2013, pp. 272–281.

[341] 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 *Proc. Int. Conf. Comput. Softw. Appl.*, 1989, pp. 527–534.

[342] S. Tallam, “Fault location and avoidance in long-running multithreaded applications,” Ph.D. dissertation, Univ. Arizona, Tucson, AZ, USA, 2007.

[343] F. Tip and T. B. Dinesh, “A slicing-based approach for locating type errors,” *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 1, pp. 5–55, Jan. 2001.

[344] F. Tip, “A survey of program slicing techniques,” *J. Program. Language*, vol. 3, no. 3, pp. 121–189, Mar. 1995.

[345] E. Torlak, M. Vaziri, and J. Dolby, “MemSAT: Checking axiomatic specifications of memory models,” in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, Toronto, ON, Canada, Jun. 2010, pp. 341–350.

[346] R. Vayani, “Improving automatic software fault localization,” Master thesis, Delft Univ. Technol., Delft, The Netherlands, 2007.

[347] I. Vessy, “Expertise in debugging computer programs: A process analysis,” *Int. J. Man-Mach. Stud.*, vol. 23, no. 5, pp. 459–494, Mar. 1985.

[348] L. Vidacs, A. Bezedes, D. Tengeri, I. Siket, and T. Gyimothy, “Test suite reduction for fault detection and localization: A combined approach,” in *Proc. IEEE Conf. Softw. Maintenance, Reeng. Reverse Eng.*, Antwerp, Belgium, Feb. 2014, pp. 204–213.

[349] L. Wang and S. D. Stoller, “Accurate and efficient runtime detection of atomicity errors in concurrent programs,” in *Proc. ACM SIGPLAN Symp. Principles Practice Parallel Program.*, Mar. 2006, pp. 137–146.

[350] S. Wang, L. David, L. Jiang, Lucia, and H. Lau, “Search-based fault localization,” in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, Lawrence, KS, USA, Nov. 2011, pp. 556–559.

[351] T. Wang, “Post-mortem dynamic analysis for software debugging,” Ph.D. dissertation, Fudan Univ., Shanghai, China, 2007.

[352] T. Wang and A. Roychoudhury, “Automated path generation for software fault localization,” in *Proc. Int. Conf. Autom. Softw. Eng.*, Long Beach, CA, USA, Nov. 2005, pp. 347–351.

[353] W. Wang, C. Wu, P. Yew, X. Yuan, Z. Wang, J. Li, and X. Feng, “Concurrency bug localization using shared memory access pairs,” in *Proc. 19th ACM SIGPLAN Symp. Principles Practices Parallel Program.*, Orlando, FL, USA, Feb. 2014, pp. 375–376.

[354] X. Wang, “Automatic localization of code omission faults,” Ph.D. dissertation, the Hong Kong Univ. Sci. Technol., Hong Kong, 2010.

[355] 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 *Proc. Int. Conf. Softw. Eng.*, May 2009, pp. 45–55.

[356] Y. Wang, H. Patil, C. Pereira, G. Lueck, R. Gupta, and I. Neamtiu, “DrDebug: Deterministic replay based cyclic debugging with dynamic slicing,” in *Proc. IEEE Int. Symp. Code Gener. Optimization*, Orlando, FL, USA, Feb. 2014, pp. 98–108.

[357] P. D. Wasserman, *Advanced Methods in Neural Computing*. Van Nostrand Reinhold, New York, 1993.

[358] D. Weeratunge, X. Zhang, W. N. Sumner, and S. Jagannathan, “Analyzing concurrency bugs using dual slicing,” in *Proc. Int. Symp. Softw. Testing Anal.*, Trento, Italy, Jul. 2010, pp. 253–264.

[359] Z. Wei and B. Han, “Multiple-bug oriented fault localization: A parameter-based combination approach,” in *Proc. 7th Int. Conf. Softw. Security Rel. Companion*, Washington, D.C., USA, Jun. 2013, pp. 125–130.

[360] M. Weiser, “Program slicing,” *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 352–357, Jul. 1984.

[361] M. Weiser, “Program slicing: formal, psychological, and practical investigations of an automatic program abstraction method,” Ph.D. dissertation, Univ. Michigan, Ann Arbor, MI, USA, 1979

[362] M. Weiglhofer, G. Fraser, and F. Wotawa, “Using spectrum-based fault localization for test case grouping,” in *Proc. Int. Conf. Autom. Softw. Eng.*, Auckland, New Zealand, Nov. 2009, pp. 630–634.

[363] W. Wen, “Software fault localization based on program slicing spectrum,” in *Proc. Int. Conf. Softw. Eng.*, Zurich, Switzerland, Jun. 2012, pp. 1511–1514.

[364] P. Willett, “Similarity-based approaches to virtual screening,” *Biochemical Soc. Trans.*, vol. 31, no. 3, pp. 603–606, Jun. 2003.

[365] C. S. Wright and T. A. Zia, “A quantitative analysis into the economics of correcting software bugs,” in *Proc. Int. Conf. Comput. Intell. Security Inf. Syst.*, Torremolinos, Spain, Jun. 2011, pp. 198–205.

[366] W. E. Wong, V. Debroy, and B. Choi, “A family of code coverage-based heuristics for effective fault localization,” *J. Syst. Softw.*, vol. 83, no. 2, pp. 188–208, Feb. 2010.

[367] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham, “Effective software fault localization using an RBF neural network,” *IEEE Trans. Rel.*, vol. 61, no. 1, pp. 149–169, Mar. 2012.

[368] W. E. Wong, V. Debroy, A. Surampudi, H. Kim, and M. F. Siok, “Recent catastrophic accidents: Investigating how software was responsible,” in *Proc. 4th Int. Conf. Secure Softw. Integr. Rel. Improvement*, 2010, pp. 14–22.

[369] W. E. Wong, V. Debroy, and D. Xu, “Towards better fault localization: A crosstab-based statistical approach,” *IEEE Trans. Syst., Man Cybern. C, Appl. Rev.*, vol. 42, no. 3, pp. 378–396, May 2012.

[370] W. E. Wong, V. Debroy, Y. Li, and R. Gao, “Software fault localization using DStar (D*),” in *Proc. 6th Int. Conf. Softw. Security Rel.*, Washington, D.C., USA, Jun. 2012, pp. 21–30.

[371] W. E. Wong, V. Debroy, R. Gao, and Y. Li, “The DStar method for effective software fault localization,” *IEEE Trans. Rel.*, vol. 63, no. 1, pp. 290–308, Mar. 2014.

[372] W. E. Wong and J. J. Li, “An integrated solution for testing and analyzing java applications in an industrial setting,” in *Proc. Asia-Pacific Softw. Eng. Conf.*, Taipei, Taiwan, Dec. 2005, pp. 576–583.

[373] W. E. Wong and Y. Qi, “Effective program debugging based on execution slices and inter-block data dependency,” *J. Syst. Softw.*, vol. 79, no. 7, pp. 891–903, Jul. 2006.

[374] W. E. Wong and Y. Qi, “BP neural network-based effective fault localization,” *Int. J. Softw. Eng. Knowl. Eng.*, vol. 19, no. 4, pp. 573–597, Jun. 2009.

[375] W. E. Wong, T. Sugeta, Y. Qi, and J. C. Maldonado, “Smart debugging software architectural design in SDL,” *J. Syst. Softw.*, vol. 76, no. 1, pp. 15–28, 2005.

[376] F. Wotawa, “Debugging hardware designs using a value-based model,” *Appl. Intell.*, vol. 16, no. 1, pp. 71–92, Jan. 2002.

[377] F. Wotawa, M. Nica, and I. Moraru, “Automated debugging based on a constraint model of the program and a test case,” *J. Logic Algebraic Program.*, vol. 81, no. 4, pp. 390–407, May 2012.

[378] F. Wotawa, “On the relationship between model-based debugging and program slicing,” *Artif. Intell.*, vol. 135, nos. 1–2, pp. 125–143, Feb. 2002.

[379] F. Wotawa, “Fault localization based on dynamic slicing and hitting-set computation,” in *Proc. Int. Conf. Quality Softw.*, Zhangjiajie, China, Jul. 2010, pp. 161–170.

[380] F. Wotawa, M. Stumptner, and W. Mayer, “Model-based debugging or how to diagnose programs automatically,” in *Proc. Int. Conf. Ind. Eng., Appl. Artif. Intell. Expert Syst.*, Cairns, Qld., Australia, Jun. 2002, pp. 746–757.

[381] F. Wotawa, J. Weber, M. Nica, and R. Ceballos, “On the complexity of program debugging using constraints for modeling the program’s syntax and semantics,” in *Proc. Conf. Spanish Assoc. Artif. Intell.*, Seville, Spain, Nov. 2009, pp. 22–31.

[382] J. Wu, X. Jia, C. Liu, H. Yang, C. Liu, and M. Jin, “A statistical model to locate faults at input levels,” in *Proc. Int. Conf. Autom. Softw. Eng.*, Linz, Austria, Sep. 2004, pp. 274–277.

[383] X. Xie, “On the analysis of spectrum-based fault localization,” Ph.D. dissertation, Swinburne Univ. Technol., Hawthorn, Vic., Australia, 2012.

[384] X. Xie, T. Y. Chen, F.-C. Kuo, and B. W. Xu, “A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization,” *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, article 31, Oct. 2013.

[385] X. Xie, T. Y. Chen, and B. Xu, “Isolating suspiciousness from spectrum-based fault localization techniques,” in *Proc. Int. Conf. Quality Softw.*, Zhangjiajie, China, Jul. 2010, pp. 385–392.

[386] X. Xie, W. E. Wong, T. Y. Chen, and B. Xu, “Spectrum-based fault localization: Testing oracles are no longer mandatory,” in *Proc. Int. Conf. Quality Softw.*, Xi’an, China, Aug. 2012, pp. 1–10.

[387] X. Xie, W. E. Wong, T. Y. Chen, and B. Xu, “Metamorphic slice: An application in spectrum-based fault localization,” *Inf. Softw. Technol.*, vol. 55, no. 5, pp. 866–879, May 2013.

[388] X. Xie, F. Kuo, T. Y. Chen, S. Yoo, and M. Harrman, “Provably optimal and human-competitive results in SBSE for spectrum based fault localization,” in *Proc. Int. Symp. Search Based Softw. Eng.*, Saint Petersburg, Russia, Aug. 2013, pp. 224–238.

[389] B. Xin, W. N. Sumner, and X. Zhang, “Efficient program execution indexing,” in *Proc. Conf. Program. Language Des. Implementation*, Tucson, AZ, USA, Jun. 2008, pp. 238–248.

[390] J. Xu, W. K. Chan, Z. Zhang, T. H. Tse, and S. Li, “A dynamic fault localization technique with noise reduction for Java programs,” in *Proc. Int. Conf. Quality Softw.*, Madrid, Spain, Jul. 2011, pp. 11–20.

[391] J. Xu, Z. Zhang, W. K. Chan, T. H. Tse, Y. T. Yu, and P. Hu, “A general noise-reduction framework for fault localization of Java program,” *Inf. Softw. Technol.*, vol. 55, no. 5, pp. 880–896, May 2013.

[392] M. Xu, R. Bodik, and M. D. Hill, “A serializability violation detector for shared-memory server programs,” in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, Chicago, IL, USA, Jun. 2005, pp. 1–14.

[393] X. Xu, V. Debroy, W. E. Wong, and D. Guo, “Ties within fault localization rankings: Exposing and addressing the problem,” *Int. J. Softw. Eng. Knowl. Eng.*, vol. 21, no. 6, pp. 803–827, Sep. 2011.

[394] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, “A brief survey of program slicing,” *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 2, pp. 1–36, Mar. 2005.

[395] X. Xue and A. S. Namin, “How significant is the effect of fault interactions on coverage-based fault localizations?” in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Measurement*, Baltimore, MD, USA, Oct. 2013, pp. 113–122.

[396] C. Yilmaz, A. Paradkar, and C. Williams, “Time will tell: Fault localization using time spectra,” in *Proc. Int. Conf. Softw. Eng.*, Leipzig, Germany, May 2008, pp. 81–90.

[397] S. Yoo, X. Xie, F. Kuo, T. Y. Chen, and M. Harman, “No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist,” Res. Note RN/14/14, Univ. College London, London, U.K., Nov. 2014.

[398] Z. You, Z. Qin, and Z. Zheng, “Statistical fault localization using execution sequence,” in *Proc. Int. Conf. Mach. Learn. Cybern.*, Xi’an, China, Jul. 2012, pp. 899–905.

[399] Y. Yu, J. Jones, and M. J. Harrold, “An empirical study of the effects of test-suite reduction on fault localization,” in *Proc. Int. Conf. Softw. Eng.*, Leipzig, Germany, May 2008, pp. 201–210.

[400] A. Zeller, “Isolating cause-effect chains from computer programs,” in *Proc. ACM SIGSOFT Symp. Found. Softw. Eng.*, Charleston, CA, USA, Nov. 2002, pp. 1–10.

[401] A. Zeller and R. Hildebrandt, “Simplifying and isolating failure-inducing input,” *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, Feb. 2002.

[402] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Amsterdam, The Netherlands: Elsevier, 2006.

[403] S. Zhang and C. Zhang, “Software bug localization with Markov logic,” in *Proc. Companion Proc. 36th Int. Conf. Softw. Eng.*, Hyderabad, India, Jun. 2014, pp. 424–427.

[404] X. Zhang, “Secure and efficient network fault localization,” Ph.D. dissertation, Carnegie Mellon Univ., Pittsburgh, PA, USA, 2012.

[405] X. Zhang, N. Gupta, and R. Gupta, “Locating faults through automated predicate switching,” in *Proc. Int. Conf. Softw. Eng.*, Shanghai, China, May 2006, pp. 272–281.

[406] X. Zhang, N. Gupta, and R. Gupta, “A study of effectiveness of dynamic slicing in locating real faults,” *Empirical Softw. Eng.*, vol. 12, no. 2, pp. 143–160, Apr. 2007.

[407] X. Zhang, N. Gupta, and R. Gupta, “Locating faulty code by multiple points slicing,” *Softw. Practice Experience*, vol. 37, no. 9, pp. 935–961, Jul. 2007.

[408] X. Zhang, R. Gupta, and Y. Zhang, “Precise dynamic slicing algorithms,” in *Proc. Int. Conf. Softw. Eng.*, Portland, OR, USA, May 2003, pp. 319–329.

[409] X. Zhang, R. Gupta, and Y. Zhang, “Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams,” in *Proc. Int. Conf. Softw. Eng.*, Edinburgh, U.K., May 2004, pp. 502–511.

[410] X. Zhang, H. He, N. Gupta, and R. Gupta, “Experimental evaluation of using dynamic slices for fault location,” in *Proc. Int. Workshop Autom. Debugging*, Monterey, CA, USA, Sep. 2005, pp. 33–42.

[411] X. Zhang, S. Tallam, N. Gupta, and R. Gupta, “Towards locating execution omission errors,” in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, San Diego, CA, USA, Jun. 2007, pp. 415–424.

[412] X. Zhang, “Fault Localization via precise dynamic slicing,” Ph.D. dissertation, Univ. Arizona, Tucson, AZ, USA, 2007.

[413] Z. Zhang, “Software debugging through dynamic analysis of program structures,” Ph.D. dissertation, The University of Hong Kong, Hong Kong, 2009.

[414] Z. Zhang, B. Jiang, W. K. Chan, and T. H. Tse, "Debugging through evaluation sequences: A controlled experimental study," in *Proc. Int. Comput. Softw. Appl. Conf.*, Turku, Finland, Jul. 2008, pp. 128–135.

[415] Z. Zhang, B. Jiang, W. K. Chan, T. H. Tse, and X. Wang, "Fault localization through evaluation sequences," *J. Syst. Softw.*, vol. 83, no. 2, pp. 174–187, Feb. 2010.

[416] Z. Zhang, W. K. Chan, T. H. Tse, Y. T. Yu, and P. Hu, "Non-parametric statistical fault localization," *J. Syst. Softw.*, vol. 84, no. 6, pp. 885–905, Jun. 2011.

[417] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang, "Capturing propagation of infected program states," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, New York, NY, USA, Aug. 2009, pp. 43–52.

[418] Z. Zhang, B. Jiang, W. K. Chan, and T. H. Tse, "Precise propagation of fault-failure correlations in program flow graphs," in *Proc. 35th Annu. Int. Comput. Softw. Appl. Conf.*, Munich, Germany, Jul. 2011, pp. 58–67.

[419] Z. Zhang, W. K. Chan, and T. H. Tse, "Fault localization based only on failed runs," *IEEE Comput.*, vol. 45, no. 6, pp. 64–71, May 2012.

[420] Z. Zhang, W. K. Chan, T. H. Tse, P. Hu, and X. Wang, "Is non-parametric hypothesis testing model robust for statistical fault localization?" *Inf. Softw. Technol.*, vol. 51, no. 11, pp. 1573–1585, Nov. 2009.

[421] L. Zhao, L. Wang, Z. Xiong, and D. Gao, "Execution-aware fault localization based on the control flow analysis," in *Proc. Int. Conf. Inf. Comput. Appl.*, Tangshan, China, Oct. 2010, pp. 158–165.

[422] L. Zhao, L. Wang and X. Yin, "Context-aware fault localization via control flow analysis," *J. Softw.*, vol. 6, no. 10, pp. 1977–1984, Oct. 2011.

[423] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: Simultaneous isolation of multiple bugs," in *Proc. Int. Conf. Mach. Learn.*, Pittsburgh, PA, USA, Jun. 2006, pp. 26–29.

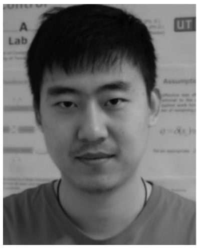
[424] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE Trans. Softw. Eng.*, vol. 32, no. 4, pp. 240–253, Apr. 2006.

[425] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed?–more accurate information retrieval-based bug localization based on bug reports," in *Proc. Int. Conf. Softw. Eng.*, Zurich, Switzerland, Jun. 2012, pp. 14–24.

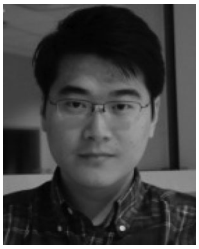
[426] T. Zimmermann and A. Zeller, "Visualizing memory graphs," in *Proc. Int. Semin. Softw. Vis.*, Apr. 2002, pp. 191–204.

[427] χSuds Users Manual, Telcordia Technologies (formerly Bellcore), Piscataway, NJ, USA, 1998.

W. Eric Wong received the MS and PhD degrees in computer science from Purdue. He is a full professor and the founding director of the Advanced Research Center for Software Testing and Quality Assurance in Computer Science, University of Texas at Dallas (UTD). He also has an appointment as a guest researcher with National Institute of Standards and Technology (NIST), an agency of the US Department of Commerce. Prior to joining UTD, he was with Telcordia Technologies (formerly Bellcore) as a senior research scientist and the project manager in charge of Dependable Telecom Software Development. In 2014, he was named the IEEE Reliability Society Engineer of the Year. His research focuses on helping practitioners improve the quality of software while reducing the cost of production. In particular, he is working on software testing, debugging, risk analysis/metrics, safety, and reliability. He has very strong experience developing real-life industry applications of his research results. He has published more than 170 papers and coedited two books. He currently serves as the vice president for Publications of the IEEE Reliability Society, and is the Founding Steering Committee Chair of the IEEE International Conference on Software Quality, Reliability, and Security (QRS). He has served as special issue guest editor for *IEEE TR*, *JSS*, *SPE*, *IST*, *SQJ*, *IJSEKE*, etc. He is also on the editorial board of the *IEEE Transactions on Reliability* and the *Journal of Systems and Software*.



Ruizhi Gao received the BS degree in software engineering from Nanjing University and the MS degree in computer science from the University of Texas at Dallas. He is currently working toward the PhD degree under the supervision of Professor W. Eric Wong at the University of Texas at Dallas. His research focus is on software testing, fault localization, and program debugging.



Yihao Li received the BS degree in software engineering from East China Institute of Technology and the MS degree in computer science from Southeastern Louisiana University. He is currently working toward the PhD degree under Professor Eric Wong's supervision in computer science at the University of Texas at Dallas. His research interests include software risk analysis, software fault localization, and program debugging.

Rui Abreu graduated in systems and computer engineering from the University of Minho, Portugal, carrying out his graduation thesis project at Siemens S.A., Portugal. Between September 2002 and February 2003, he followed courses of the Software Technology Master Course at the University of Utrecht, The Netherlands, as an Erasmus Exchange student. He was an intern researcher at Philips Research Labs, the Netherlands, between October 2004 and June 2005, and received the PhD degree from the Delft University of Technology, The Netherlands, in November 2009. He is currently an assistant professor at the Faculty of Engineering, University of Porto, Portugal. He has also been an applied research scientist with Palo Alto Research Center, USA, since July 2014.

Franz Wotawa received the MSc degree in computer science in 1994 and the PhD degree in 1996, both from the Vienna University of Technology. He is currently a professor of software engineering at the Graz University of Technology. His research interests include model-based and qualitative reasoning, theorem proving, mobile robots, verification and validation, and software testing and debugging. Besides theoretical foundations, he has always been interested in closing the gap between research and practice. For this purpose he founded Softnet Austria in 2006, which is a nonprofit organization carrying out applied research projects together with companies. He has written more than 290 papers for journals, books, conferences, and workshops. He has supervised 71 MS and 27 PhD students. He has been a member of a number of program committees and organized workshops and special issues of journals. He is a member of the Academia Europaea, the IEEE Computer Society, ACM, the Austrian Computer Society (OCG), and the Austrian Society for Artificial Intelligence and a senior member of the AAAI.