1

Software Fault Localization: an Overview of Research, Techniques, and Tools

W. Eric Wong[1], Ruizhi Gao[2], Yihao Li[3], Rui Abreu[4], Franz Wotawa[5], and Dongcheng Li[1]

[1] *Department of Computer Science, University of Texas at Dallas, Richardson, TX, USA*
[2] *Sonos Inc., Boston, MA, USA*
[3] *School of Information and Electrical Engineering, Ludong University, Yantai, China*
[4] *Department of Informatics Engineering, Faculty of Engineering, University of Porto, Porto, Portugal*
[5] *Institute of Software Technology, Graz University of Technology, Graz, Austria*

1.1 Introduction

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 book, we categorize and provide a comprehensive overview of such techniques and discuss key issues and concerns that are pertinent to software fault localization.

Software is fundamental to our lives today, and with its ever-increasing usage and adoption, its influence is practically ubiquitous. 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 chapter is an extension of Wong, W.E., Gao, R., Li, Y., Abreu, R., and Wotawa, F. (2016). A survey on software fault localization. *IEEE Transactions on Software Engineering* 42 (8): 707–740.

Handbook of Software Fault Localization: Foundations and Advances, First Edition.
Edited by W. Eric Wong and T.H. Tse.

Published 2023 by John Wiley & Sons, Inc.

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 [1–3]. On 15 January 1990, the AT&T operation center in Bedminster, NJ, USA, had an increase of red warning signals appearing across the 75 screens that indicated the status of parts of the AT&T worldwide network. As a result, only about 50 percent of the calls made through AT&T were connected. It took nine hours for the AT&T technicians to identify and fix the issue caused by a misplaced break statement in the code. AT&T lost $60 to $75 million in this accident [4].

Furthermore, software faults in safety-critical systems have significant ramifications not only limited to financial loss, but also to loss of life, which is alarming [5]. On 20 December 1995, a Boeing 757 departed from Miami, FL, USA. The aircraft was heading to Cali, Colombia. However, it crashed into a 9800 feet mountain. A total of 159 deaths resulted; leaving only five passengers alive. This event marked the highest death toll of any accident in Colombia at the time. This accident was caused by the inconsistencies between the naming conventions of the navigational charts and the flight management system. When the crew looked up the waypoint "Rozo", the chart indicated the letter "R" as its identifier. The flight management system, however, had the city paired with the word "Rozo". As a result, when the pilot entered the letter "R", the system did not know if the desired city was Rozo or Romeo. It automatically picked Romeo, which is a larger city than Rozo, as the next waypoint.

A 2006 report from the National Institute of Standards and Technology (NIST) [6] indicated that software errors are estimated to cost the US 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 [7], 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 state-of-the-art techniques in fault localization in order to offer a comprehensive resource for those already in the area and 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, in this chapter, we created a publication repository that includes 587 papers published from 1977 to 2020. We also searched for Masters' and PhD theses closely related to software fault localization, which are listed in Table 1.1.

Table 1.1 A list of recent PhD and Masters' theses on software fault localization.

Author	Title	Degree	University	Year
Ehud Y. Shapiro [8]	Algorithmic Program Debugging	PhD	Yale University	1983
Hiralal Agrawal [9]	Towards Automatic Debugging of Computer Programs	PhD	Purdue University	1991
Hsin Pan [10]	Software debugging with dynamic instrumentation and test-based knowledge	PhD	Purdue University	1993
W. Bond Gregory [11]	Logic Programs for Consistency-based Diagnosis	PhD	Carleton University	1994
Benjamin Robert Liblit [12]	Cooperative Bug Isolation	PhD	The University of California, Berkeley	2004
Bernhard Peischl [13]	Automated Source-Level Debugging of Synthesizeable VHDL Designs	PhD	Graz University of Technology	2004
Haifeng He [14]	Automated Debugging using Path-based Weakest Preconditions	Master	University of Arizona	2004
Alex David Groce [15]	Error Explanation and Fault Localization with Distance Metrics	PhD	Carnegie Mellon University	2005

(*Continued*)

Table 1.1 (Continued)

Author	Title	Degree	University	Year
Emmanuel Renieris [16]	A Research Framework for Software-Fault Localization Tools	PhD	Brown University	2005
Daniel Köb [17]	Extended Modeling for Automatic Fault Localization in Object-Oriented Software	PhD	Graz University of Technology	2005
David Hovemeyer [18]	Simple and Effective Static Analysis to Find Bugs	PhD	University of Maryland	2005
Peifeng Hu [19]	Automated Fault Localization: a Statistical Predicate Analysis Approach	PhD	The University of Hong Kong	2006
Xiangyu Zhang [20]	Fault Localization via Precise Dynamic Slicing	PhD	The University of Arizona	2006
Rafi Vayani [21]	Improving Automatic Software Fault Localization	Master	Delft University of Technology	2007
Ramana Rao Kompella [22]	Fault Localization in Backbone Networks	PhD	University of California, San Diego	2007
Andreas Griesmayer [23]	Debugging Software: from Verification to Repair	PhD	Graz University of Technology	2007
Tao Wang [24]	Post-Mortem Dynamic Analysis For Software Debugging	PhD	Fudan University	2007
Sriraman Tallam [25]	Fault Location and Avoidance in Long-Running Multithreaded Applications	PhD	The University of Arizona	2007
Ophelia C. Chesley [26]	CRISP-A fault localization Tool for Java Programs	Master	Rutgers, The State University of New Jersey	2007
Shan Lu [27]	Understanding, Detecting and Exposing Concurrency Bugs	PhD	University of Illinois at Urbana-Champaign	2008

Table 1.1 (Continued)

Author	Title	Degree	University	Year
Naveed Riaz [28]	Automated Source-Level Debugging of Synthesizable Verilog Designs	PhD	Graz University of Technology	2008
James Arthur Jones [29]	Semi-Automatic Fault Localization	PhD	Georgia Institute of Technology	2008
Zhenyu Zhang [30]	Software Debugging through Dynamic Analysis of Program Structures	PhD	The University of Hong Kong	2009
Rui Abreu [31]	Spectrum-based Fault Localization in Embedded Software	PhD	Delft University of Technology	2009
Dennis Jefferey [32]	Dynamic State Alteration Techniques for Automatically Locating Software Errors	PhD	University of California Riverside	2009
Xinming Wang [33]	Automatic Localization of Code Omission Faults	PhD	The Hong Kong University of Science and Technology	2010
Fabrizio Pastore [34]	Automatic Diagnosis of Software Functional Faults by Means of Inferred Behavioral Models	PhD	University of Milan Bicocca	2010
Mihai Nica [35]	On the Use of Constraints in Automated Program Debugging – From Foundations to Empirical Results	PhD	Graz University of Technology	2010
Zachary P. Fry [36]	Fault Localization Using Textual Similarities	Master	The University of Virginia	2011
Hua Jie Lee [37]	Software Debugging Using Program Spectra	PhD	The University of Melbourne	2011

(*Continued*)

Table 1.1 (Continued)

Author	Title	Degree	University	Year
Vidroha Debroy [38]	Towards the Automation of Program Debugging	PhD	The University of Texas at Dallas	2011
Alberto Gonzalez Sanchez [39]	Cost Optimizations in Runtime Testing and Diagnosis	PhD	Delft University of Technology	2011
Jared David DeMott [40]	Enhancing Automated Fault Discovery and Analysis	PhD	Michigan State University	2012
Xin Zhang [41]	Secure and Efficient Network Fault Localization	PhD	Carnegie Mellon University	2012
Xiaoyuan Xie [42]	On the Analysis of Spectrum-based Fault Localization	PhD	Swinburne University of Technology	2012
Alexandre Perez [43]	Dynamic Code Coverage with Progressive Detail Levels	Master	University of Porto	2012
Raul Santelices [44]	Change-effects Analysis for Effective Testing and Validation of Evolving Software	PhD	Georgia Institute of Technology	2012
George. K. Baah [45]	Statistical Causal Analysis for Fault Localization	PhD	Georgia Institute of Technology	2012
Swarup K. Sahoo [46]	A Novel Invariants-based Approach for Automated Software Fault Localization	PhD	University of Illinois at Urbana-Champaign	2012
Birgit Hofer [47]	From Fault Localization of Programs written in 3rd level Language to Spreadsheets	PhD	Graz University of Technology	2013
Aritra Bandyopadhyay [48]	Mitigating the Effect of Coincidental Correctness in Spectrum-based Fault Localization	PhD	Colorado State University	2013

Table 1.1 (Continued)

Author	Title	Degree	University	Year
Shounak Roychowdhury [49]	A Mixed Approach to Spectrum-based Fault Localization Using Information Theoretic Foundations	PhD	The University of Texas at Austin	2013
Shaimaa Ali [50]	Localizing State-Dependent Faults Using Associated Sequence Mining	PhD	The University of Western Ontario	2013
Christian Kuhnert [51]	Data-driven Methods for Fault Localization in Process Technology	PhD	Karlsruhe Institute of Technology	2013
Dawei Qi [52]	Semantic Analyses to Detect and Localize Software Regression Errors	PhD	Tsinghua University	2013
William N. Sumner [53]	Automated Failure Explanation Through Execution Comparison	PhD	Purdue University	2013
Mark A. Hays [54]	A Fault-based Model of Fault Localization Techniques	PhD	University of Kentucky	2014
Sang Min Park [55]	Effective Fault Localization Techniques for Concurrent Software	PhD	Georgia Institute of Technology	2014
Gang Shu [56]	Statistical Estimation of Software Reliability and Failure-causing Effect	PhD	Case Western Reserve University	2014
Lucia [57]	Ranking-based Approaches for Localizing Faults	PhD	Singapore Management University	2014
Seok-Hyeon Moon [58]	Effective Software Fault Localization using Dynamic Program Behaviors	Master	Korea Advanced Institute of Science and Technology	2014

(*Continued*)

Table 1.1 (Continued)

Author	Title	Degree	University	Year
Yepang Liu [59]	Automated Analysis of Energy Efficiency and Performance for Mobile Applications	PhD	The Hong Kong University of Science and Technology	2014
Cuiting Chen [60]	Automated Fault Localization for Service-Oriented Software Systems	PhD	Delft University of Technology	2015
Matthias Rohr [61]	Workload-sensitive Timing Behavior Analysis for Fault Localization in Software Systems	PhD	Kiel University	2015
Ozkan Bayraktar [62]	Ela: an Automated Statistical Fault Localization Technique	PhD	The Middle East Technical University	2015
Azim Tonzirul [63]	Fault Discovery, Localization, and Recovery in Smartphone Apps	PhD	University of California Riverside	2016
Laleh Gholamosseinghandehari [64]	Fault Localization based on Combinatorial Testing	PhD	The University of Texas at Arlington	2016
Ruizhi Gao [65]	Advanced Software Fault Localization for Programs with Multiple Bugs	PhD	The University of Texas at Dallas	2017
Shih-Feng Sun [66]	Statistical Fault Localization and Causal Interactions	PhD	Case Western Reserve University	2017
Rongxin Wu [67]	Automated Techniques for Diagnosing Crashing Bugs	PhD	The Hong Kong University of Science and Technology	2017
Arjun Roy [68]	Simplifying dataleft fault detection and localization	PhD	University of California San Diego	2018

Table 1.1 (Continued)

Author	Title	Degree	University	Year
Yun Guo [69]	Towards Automatically Localizing and Repairing SQL Faults	PhD	George Mason University	2018
Nasir Safdari [70]	Learning to Rank Relevant Files for Bug Reports Using Domain knowledge, Replication and Extension of a Learning-to-Rank Approach	Master	Rochester Institute of Technology	2018
Dai Ting [71]	A Hybrid Approach to Cloud System Performance Bug Detection, Diagnosis and Fix	PhD	North Carolina State University	2019
George Thompson [72]	Towards Automated Fault Localization for Prolog	Master	North Carolina A&T State University	2020
Xia Li [73]	An Integrated Approach for Automated Software Debugging via Machine Learning and Big Code Mining	PhD	The University of Texas at Dallas	2020
Muhammad Ali Gulzar [74]	Automated Testing and Debugging for Big Data Analytics	PhD	University of California, Los Angeles	2020
Mihir Mathur [75]	Leveraging Distributed Tracing and Container Cloning for Replay Debugging of Microservices	Master	University of California, Los Angeles	2020

All papers in our repository[2] are sorted by year, and the result is displayed in Figure 1.1. 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 two decades.

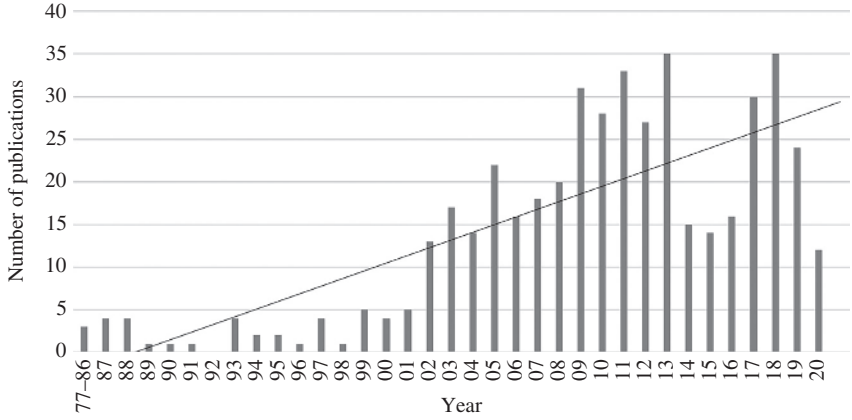

Figure 1.1 Papers on software fault localization from 1977 to 2020.

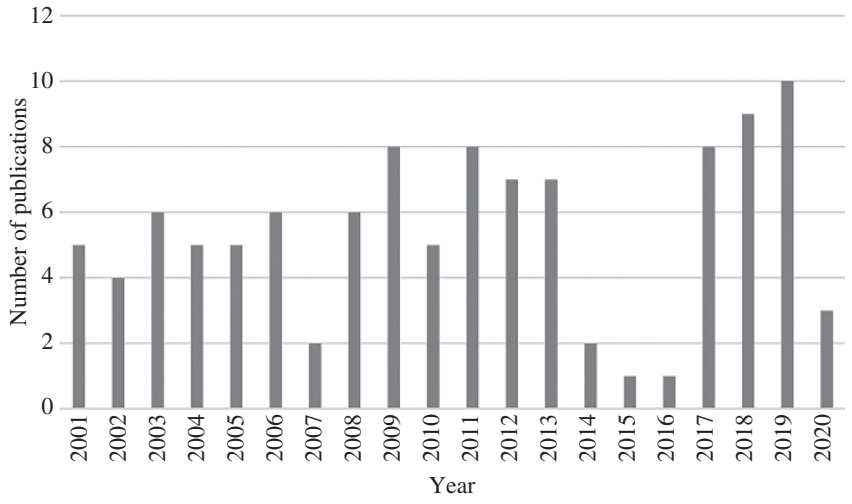

Figure 1.2 Publications on software fault localization in top venues from 2001 to 2020.

Also, as per our repository, Figure 1.2. gives the number of publications related to software fault localization that have 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 2019. 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 two decades.

There is thus a rich collection of literature on various techniques that aim to facilitate fault localization and make it more effective. 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 book, our primary focus is on the techniques for locating Bohrbugs [76]. Those for diagnosing Mandelbugs [76] 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 chapter, and thus for convenience, we provide definitions for them here per the taxonomy provided in [77]:

- 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.

In this book, we group fault localization techniques into appropriate categories (including traditional, slicing-based, spectrum-based, statistics-based, machine learning-based, data mining-based, information-retrieval-based, model-based, spreadsheet-based, and emerging techniques) for collective discussion with an emphasis on the most important features. We introduce 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 also described as well as a discussion of fault localization tools and theoretical studies. Moreover, we explore some critical aspects of software fault localization, including (i) fault

localization for programs with multiple bugs, (ii) inputs, outputs, and impact of test cases, (iii) coincidental correctness, (iv) faults introduced by missing code, (v) combination of multiple fault localization techniques, (vi) ties within fault localization rankings, and (vii) fault localization for concurrency bugs. The general information of each chapter is introduced as follows.

This book begins by introducing traditional software fault localization techniques in Chapter 2, including program logging, assertions, and breakpoints. Examples will also be provided to clearly explain these 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 in the same way as the original program with respect to certain specifications. Chapter 3 introduces slicing-based fault localization techniques, which can be classified into three major categories: static slicing, dynamic slicing, and execution slicing-based techniques. Examples will be given to illustrate the differences among these categories. Techniques based on other slicing such as dual slicing, thin slicing, and relevant slicing are also included.

Program spectrum-based techniques are presented in Chapter 4. A program spectrum details the execution information of a program from certain perspectives, such as execution information for conditional branches or loop-free intraprocedural paths. It can be used to track program behavior. A list of different kinds of program spectra will be provided. Also discussed are issues and concerns related to program spectrum-based techniques.

Software fault localization techniques based on well-defined statistical analyses (e.g. parametric and nonparametric hypothesis testing, causal-inference analysis, and cross tabulation analysis) are described in Chapter 5.

Machine learning is the study of computer algorithms that improve through experience. These techniques are adaptive and robust and can produce models based on data, with limited human interaction. Such properties have led to their employment in many disciplines including bioinformatics, natural language processing, cryptography, computer vision, etc. In the context of software 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. Chapter 6 covers fault localization techniques based on machine learning 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. Data mining-based techniques are reviewed and analyzed in Chapter 7.

Chapter 8 introduces information retrieval (IR)-based fault localization techniques. Fault localization is the problem of identifying buggy source code files given a textual description of a bug. This problem is important since many bugs are reported through bug tracking systems like Bugzilla and Jira, and the number of bug reports is often too many for developers to handle. This necessitates an automated tool that can help developers identify relevant files given a bug report. Due to the textual nature of bug reports, IR techniques are often employed to solve this problem. Many IR-based fault localization techniques have been proposed in the literature.

Program models can be used for software fault localization. The first part of Chapter 9 discusses techniques based on different program models such as dependency-based models, abstraction-based models, and value-based models. The second part emphasizes model checking-based techniques.

Spreadsheets are one of the most popular types of end-user software and have been used in many sectors, especially in business. Chapter 10 discusses how techniques using value-based or dependency-based models can effectively locate bugs in cells with erroneous formulae and avoid incorrect computation.

Instead of being evaluated empirically, the effectiveness of software fault localization techniques can also be analyzed from theoretical perspectives. Chapter 11 discusses theoretical studies on software fault localization.

Many of the software fault localization techniques assume that there is only one bug in the program under study. This assumption may not be realistic in practice. Mixed failed test cases associated with different causative bugs may reduce the fault localization effectiveness. In Chapter 12, we present fault localization techniques for programs with multiple bugs.

Finally, Chapter 13 presents emerging aspects of software fault localization, including how to apply the scientific method to fault localization, how to locate faults when the oracle is not available, how to automatically predict fault localization effectiveness, and how to integrate fault localization into automatic test generation tools.

The remaining part of this Chapter is organized in the following manner: we begin by describing traditional and intuitive fault localization techniques in Section 1.2, moving on to more advanced and complex techniques in Section 1.3. In Section 1.4, 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 1.5, followed by a discussion of fault localization tools in Section 1.6. Finally, critical aspects and conclusions are presented in Section 1.7 and Section 1.8, respectively.

1.2 Traditional Fault Localization Techniques

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

1.2.1 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 [78]. 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.

1.2.2 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 [79, 80].

1.2.3 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. [81, 82]) 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 [83] and Microsoft Visual Studio Debugger [84].

1.2.4 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. [85]).
- Identifying memory leaks or code that performs unexpectedly poorly (e.g. [86]).
- Examining the side effects of lazy evaluation (e.g. [87]).

Tools that use profiling for program debugging include GNU's gprof [88] and the Eclipse plugin TPTP [89].

1.3 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* [90, 91], 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 [91] 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 chapter, we classify fault localization techniques into nine categories, including slicing-based, spectrum-based, statistics-based, machine learning-based, data mining-based, IR-based, model-based, spreadsheet-based techniques, and additional emerging techniques. Many studies that evaluate the effectiveness of specific fault localization techniques have been reported [92–124]. However, none of them offer a comprehensive discussion on all these techniques.

1.3.1 Slicing-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 [125–127] since Weiser first proposed *static slicing* in 1979 [128].

One of the important applications of static slicing [129] 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 [130]. Although static slice-based techniques have been experimentally evaluated and confirmed to be useful in fault localization [109], 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 [131]. 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 [132], and type-checkers [133].

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 deal with the imprecision of static slicing, Zhang and Santelices [134] propose PRIOSLICE to refine the results reported by static slicing.

A good approach to exclude such extra statements from a dice (as well as a slice) is to use *dynamic slicing* [135, 136] 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 [121, 122, 132, 134, 137–159], which use the dynamic slicing concept in program debugging, have been reported. In [156], 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 [160]. Zhang et al. [157] 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 [152] 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. Ocariza et al. [150] propose an automated technique based on dynamic backward slicing of the web application to localize DOM-related JavaScript faults. The proposed fault localization approach is implemented in a tool called AUTOFLOX. Ishii and Kutsuna [143] propose an effective fault localization method for Simulink model. They use the satisfiability modulo theories (SMT) solver to generate distinct dynamic slicing result for each failed test case. Guo et al. [161] apply dynamic slicing and delta debugging to localize faults in SQL predicates. First, in order to identify any suspicious clause, row-based dynamic slicing execute the query for each row in the provided test data, and record the predicate and the Boolean result for each clause. Second, delta debugging is employed to mutate the column values of the failed rows and replace them with the corresponding values from the successful rows. If the mutated row passes, then the clause containing this column is faulty.

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 [162]. Gyimothy et al. [163] 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. [162] 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. [164] to identify root causes of omission errors in concurrent programs, in which *dual slicing*, a combination of dynamic slicing and trace differencing, is used. Wang and Liu [165] propose a hierarchical multiple

predicate switching technique (HMPS). It reduces the search scope of critical predicates to highly suspicious functions identified by spectrum-based fault localization (SBFL) techniques, and then assigns the functions into combinations following the call dependency graph.

An alternative approach to static and dynamic slicing is the use of *execution slicing* based on data-flow tests to locate program bugs [166] 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 [167–170] 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) [171, 172] and eXVantage at Avaya [173]. Agrawal et al. [166] 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. [174, 175] and Wong et al. [176] 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 code in column 2 of Table 1.2 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}, s_{13} as shown in the fifth column of Table 1.2.

Table 1.2 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 * i;` `// bug: product = product * 2 * i`	`product = product * i;`	`product = product * i;`	`product = product * 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);`

Source: Wong et al. [177]/IEEE.

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 [178]. The former includes additional code in the search domain for inspection based on its inter-block data dependency with the code 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 [179], the notion of using *barriers* is proposed to provide a filtering approach for smaller program slices and better comprehensibility. Stridharan et al. [180] 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.

1.3.2 Program Spectrum-Based Techniques

Following the discussion in the beginning of Section 1.3, 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 [181]. It can be used to track program behavior [182]. An early study by Collofello and Cousins [183] 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. Masri [184] presents a comprehensive survey of state-of-the-art SBFL techniques proposed from 2005 to February 2016, describing the most recent advances and challenges.

1.3.2.1 Notation

p	a program
a_{ef}	Number of failed test cases that cover a statement
a_{nf}	Number of failed test cases that do not cover a statement

a_{es} Number of successful test cases that cover a statement
a_{ns} Number of successful test cases that do not cover a statement
a_e Total number of test cases that cover a statement
a_n Total number of test cases that do not cover a statement
a_S Total number of successful test cases
a_f Total number of failed test cases
t_i The ith test case

1.3.2.2 Techniques

Early studies [145, 185–187] only use failed test cases for SBFL, though this approach has subsequently been deemed ineffective [107, 117, 166]. 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 [188]. 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 [188] propose 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 unchecked 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 [189], 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 [190–192].

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 [174], which uses the coverage and execution result (success or failure) to compute the suspiciousness of each statement as $\left(a_{ef}/a_f\right)/\left(a_{ef}/a_f + a_{es}/a_s\right)$. A study on the *Siemens* suite [107] 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 [193]. Based on the suspiciousness computed by Tarantula, studies like [174, 175] 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 [103], 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 us use the code in Table 1.2 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 1.3.

The third to fifth columns in Table 1.3 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 a_{ef} and a_{es} 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 [94] is generally considered more effective than Tarantula, and its formula is as follows:

$$\text{Suspiciousness(Ochiai)} = \frac{a_{ef}}{\sqrt{a_f \times (a_{ef} + a_{es})}}$$

There are two major differences between Ochiai and the nearest neighbor model: (i) the nearest neighbor model utilizes a single failed test case, while Ochiai uses multiple failed test cases and (ii) 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 [113] is an extension of Ochiai, and its formula is as follows:

$$\text{Suspiciousness(Ochiai2)} = \frac{a_{ef} \times a_{ns}}{\sqrt{(a_{ef} + a_{es}) \times (a_{ns} + a_{nf}) \times (a_{ef} + a_{nf}) \times (a_{ep} + a_{ns})}}$$

Table 1.3 An example showing the suspiciousness values computed by Tarantula.

Code with a bug at s_7	$a = 0$	$a = 1$	$a = 2$	a_{ef}	a_{es}	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 * 2 * i`			●	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} `}`	●	●	●	1	2	0.5	3
s_{13} `print (product);`	●	●	●	1	2	0.5	3
Execution result	Successful	Successful	Failed				

Source: Wong et al. [177]/IEEE.

In [113], Naish et al. propose two techniques, O and O^P. The technique O is designed for programs with a single bug, while O^P is better applied to 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. [194] present a different view by showing that Ochiai can be more effective than O and O^P for programs with single bugs.

$$\text{Suspiciousness}(O) = \begin{cases} -1 \text{ if } a_{nf} > 0 \\ a_{ns} \text{ otherwise} \end{cases}$$

Table 1.4 lists 31 similarity coefficient-based techniques, along with their algebraic forms, which have been used in different studies such as [195–197]. A few additional techniques using similar approaches can be found in [198]. Tools like Zoltar [199] and DEPUTO [200] are available to compute the suspiciousness with respect to selected techniques. Empirical studies have also shown that techniques proposed in [117, 197, 201–203] are, in general, more effective than Tarantula.

Comparisons among different SBFL techniques are frequently discussed in recent studies [95, 113, 194, 197, 204, 205]. 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 [206].

A few additional examples of program SBFL techniques are listed below.

- ***Program Invariants Hit Spectrum (PIHS)-Based:*** This spectrum records the coverage of program invariants [207], 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* [208], also called *likely invariants* [209], 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 [210], 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. Shu et al. [211] propose FLSF technique based on Tarantula. They use statement frequency, instead of coverage information, to evaluate the suspiciousness of each statement. FLSF first counts the statement frequency of each program statement in each test case execution so as to construct a statement frequency matrix. Second, each weighted element of the constructed matrix is normalized to a value between 0 and 1. Finally, the suspiciousness of each statement

Table 1.4 Some similarity coefficients used for fault localization.

Coefficient		Algebraic form	Coefficient		Algebraic form
1	Braun-Banquet	$\frac{a_{ef}}{\max\left(a_{ef}+a_{es}, a_{ef}+a_{nf}\right)}$	17	Harmonic Mean	$\frac{\left(a_{ef}\times a_{ns}-a_{nf}\times a_{es}\right)\times\left(\left(a_{ef}+a_{es}\right)\times\left(a_{ns}+a_{nf}\right)+\left(a_{ef}+a_{nf}\right)\times\left(a_{es}+a_{ns}\right)\right)}{\left(a_{ef}+a_{es}\right)\times\left(a_{ns}+a_{nf}\right)\times\left(a_{ef}+a_{nf}\right)\times\left(a_{es}+a_{ns}\right)}$
2	Dennis	$\frac{\left(a_{ef}\times a_{ns}\right)-\left(a_{es}\times a_{nf}\right)}{\sqrt{n\times\left(a_{ef}+a_{es}\right)\times\left(a_{ef}+a_{nf}\right)}}$	18	Rogot2	$\frac{1}{4}\left(\frac{a_{ef}}{a_{ef}+a_{es}}+\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ns}}{a_{ns}+a_{es}}+\frac{a_{ns}}{a_{ns}+a_{nf}}\right)$
3	Mountford	$\frac{a_{ef}}{0.5\times\left(\left(a_{ef}\times a_{es}\right)+\left(a_{ef}\times a_{nf}\right)\right)+\left(a_{es}\times a_{nf}\right)}$	19	Simple Matching	$\frac{a_{ef}+a_{ns}}{a_{ef}+a_{es}+a_{ns}+a_{nf}}$
4	Fossum	$\frac{n\times\left(a_{ef}-0.5\right)^2}{\left(a_{ef}+a_{es}\right)\times\left(a_{ef}+a_{nf}\right)}$	20	Rogers and Tanimoto	$\frac{a_{ef}+a_{ns}}{a_{ef}+a_{ns}+2\left(a_{nf}+a_{es}\right)}$
5	Pearson	$\frac{n\times\left(\left(a_{ef}\times a_{ns}\right)-\left(a_{es}\times a_{nf}\right)\right)^2}{\Phi_e\times\Phi_n\times\Phi_S\times\Phi_F}$	21	Hamming	$a_{ef}+a_{ns}$
6	Gower	$\frac{a_{ef}\times a_{ns}}{\sqrt{\Phi_F\times\Phi_e\times\Phi_n\times\Phi_S}}$	22	Hamann	$\frac{a_{ef}+a_{ns}-a_{nf}-a_{es}}{a_{ef}+a_{nf}+a_{es}+a_{ns}}$
7	Michael	$\frac{4\times\left(\left(a_{ef}\times a_{ns}\right)-\left(a_{es}\times a_{nf}\right)\right)}{\left(a_{ef}\times a_{ns}\right)^2+\left(a_{es}\times a_{nf}\right)^2}$	23	Sokal	$\frac{2\left(a_{ef}+a_{ns}\right)}{2\left(a_{ef}+a_{ns}\right)+a_{nf}+a_{es}}$
8	Pierce	$\frac{\left(a_{ef}\times a_{nf}\right)+\left(a_{nf}\times a_{es}\right)}{\left(a_{ef}\times a_{nf}\right)+\left(2\times\left(a_{nf}\times a_{ns}\right)\right)+\left(a_{es}\times a_{ns}\right)}$	24	Scott	$\frac{4\left(a_{ef}\times a_{ns}-a_{nf}\times a_{es}\right)-\left(a_{nf}-a_{es}\right)^2}{\left(2a_{ef}+a_{nf}+a_{es}\right)\left(2a_{ns}+a_{nf}+a_{es}\right)}$
9	Baroni-Urbani and Buser	$\frac{\sqrt{\left(a_{ef}\times a_{ns}\right)}+a_{ef}}{\sqrt{\left(a_{ef}\times a_{ns}\right)}+a_{ef}+a_{es}+a_{nf}}$	25	Rogot1	$\frac{1}{2}\left(\frac{a_{ef}}{2a_{ef}+a_{nf}+a_{es}}+\frac{a_{ns}}{2a_{ns}+a_{nf}+a_{es}}\right)$
10	Tarwid	$\frac{\left(n\times a_{ef}\right)-\left(\Phi_F\times\Phi_e\right)}{\left(n\times a_{ef}\right)+\left(\Phi_F\times\Phi_e\right)}$	26	Kulczynski	$\frac{a_{ef}}{a_{nf}+a_{es}}$

(*Continued*)

Table 1.4 (Continued)

Coefficient		Algebraic form	Coefficient		Algebraic form
11	Ample	$\left\lvert \frac{a_{ef}}{(a_{ef}+a_{nf})} - \frac{a_{es}}{(a_{es}+a_{ns})} \right\rvert$	27	Anderberg	$\frac{a_{ef}}{a_{ef}+2(a_{nf}+a_{es})}$
12	Phi (Geometric Mean)	$\frac{a_{ef}\times a_{ns} - a_{nf}\times a_{es}}{\sqrt{(a_{ef}+a_{es})\times(a_{ef}+a_{nf})\times(a_{es}+a_{ns})\times(a_{nf}+a_{ns})}}$	28	Dice	$\frac{2a_{ef}}{a_{ef}+a_{nf}+a_{es}}$
13	Arithmetic Mean	$\frac{2(a_{ef}\times a_{ns} - a_{nf}\times a_{es})}{(a_{ef}+a_{ns})\times(a_{ns}+a_{nf}) + (a_{ef}+a_{nf})\times(a_{es}+a_{ns})}$	29	Goodman	$\frac{2a_{ef}-a_{nf}-a_{es}}{2a_{ef}+a_{nf}+a_{es}}$
14	Cohen	$\frac{2(a_{ef}\times a_{ns} - a_{nf}\times a_{es})}{(a_{ef}+a_{ns})\times(a_{ns}+a_{es}) + (a_{ef}+a_{nf})\times(a_{nf}+a_{ns})}$	30	Jaccard	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{es}}$
15	Fleiss	$\frac{4(a_{ef}\times a_{ns} - a_{nf}\times a_{es}) - (a_{nf}-a_{es})^2}{(2a_{ef}+a_{nf}+a_{es}) + (2a_{ns}+a_{nf}+a_{es})}$	31	Sorensen-Dice	$\frac{2a_{ef}}{2a_{ef}+a_{nf}+a_{es}}$
16	Zoltar	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{es}+\frac{10000\times a_{nf}\times a_{es}}{a_{ef}}}$			

Source: Wong et al. [177]/IEEE.

is computed according to its frequency value. The proposed technique is evaluated using Siemens suite and is reported to be better than Tarantula. Le et al. [212] propose a fault localization technique that employs a learning-to-rank strategy, using likely program invariants and suspiciousness scores as features, to rank program methods based on their likelihood of being a root cause of a failure.

- ***Predicate Count Spectrum (PRCS)-Based:*** PRCS records how predicates are executed and can be used to track program behavior that is likely to be erroneous. These techniques are often labeled as *statistical debugging* techniques because the PRCS information is analyzed using statistical methods. Fault localization techniques in this category include Liblit05 [213], SOBER [214], etc. See Section 1.3.3 for more details. Naish et al. [112] 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. [215] 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. [216] 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 [217–219] 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 1.5 [181] can also be applied to identify suspicious code in a program.

1.3.2.3 Issues and Concerns

A variety of issues and concerns about SBFL 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 [220], 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.

Table 1.5 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

It is possible, however, that successful test cases may also contain bugs. In [117], 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 [178] and Guo et al. [221] 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 [222, 223]. In [120], 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 1.7.6. 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. [224, 225] 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. Chen et al. [226] use metamorphic relations and symbolic evaluation to integrate program proving, testing, and debugging. See Chapter 11 of the Handbook. Tolksdorf et al. [227] apply metamorphic test cases to interactive debuggers. They transform both the debugged code and the debugging actions in a way that the behavior of the original and the transformed inputs can differ only in specific ways.

Zhao et al. [228, 229] 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.

Guo et al. [230] discuss the instability of SBFL techniques. They provide a stochastic technique to measure the instability quantity of SBFL. Then, the necessity of evaluating SBFL instability is proven by experimental studies. Finally, the authors propose several factors such as the test suite size and the risk evaluation formula to measure the instability of SBFL. Keller et al. [231] find that SBFL has limitation in locating bugs on large-size benchmarks. Studies in [232–235] report that combining SBFL with mutation-based fault localization (MBFL) [236, 237] can improve FL effectiveness for real faults compared to using SBFL or MBFL alone. Li et al. [238] assign weights to the traditional binary executions in SBFL using the probabilities of branch executions. Instead of using 0 or 1 to represent the execution information of each statement in a test case, the value is replaced by the ratio of the number of executions of a statement located in a branch of a module to the total number of executions of the module.

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. [239] 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.

1.3.3 Statistics-Based Techniques

A statistical debugging technique (Liblit05) that can isolate bugs in programs with instrumented predicates at particular points is presented in [213]. 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. [240] 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 [214], 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. They compute $\pi(P) = n(t)/(n(t) + n(f))$, 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. [241] use a similar heuristic to rank all predicates. In addition, they apply nonparametric 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 [123, 242].

The study in [202] 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 among crosstab, SOBER, and Liblit05 is that crosstab can be generally applied to rank suspicious program elements (i.e. statement, predicate, and function/method), 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 [107], 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 PDG, 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 [202] suggest that crosstab is almost always more effective in locating bugs in the Siemens suite than Liblit05 and SOBER. Similar to crosstab [202], Yang et al. [243] use conditional probability to quantify the dependence between program spectra and execution results, and compute the suspiciousness score accordingly. Henderson and Podgurski [244] randomly sample suspicious subgraphs of dynamic control flow graphs of successful and failed executions. Metrics used in coverage-based SBFL are adapted to select the most suspicious subgraphs.

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. [245, 246] 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. [247] 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. Baah et al. [248] 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 [249], they further enhance the linear model toward better fault localization effectiveness by including information on data-flow dependence. In [250], 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. Wang et al. [251] propose a variable type-based predicate designation (VTPD) approach to improve the ability of fault-relevant predicate identification and mitigate the confounding effect. The approach begins with designing two kinds of predicates: the original type for branch statements and indeed type for assignments and return statements. The indeed type is further broken into several variable types based on the programming language, e.g. there are numeric, Boolean, character, and reference types in Java. Then, the authors

conduct a dependency analysis technique using the casual graph model to examine the potential confounding effect of control and data dependences. Finally, based on the analysis result, the authors build a linear regression model to estimate the suspiciousness of predicates by calculating the contribution to the failure result, and the control and data dependencies. Sun and Podgurski [205] analyze several coverage-based statistical fault localization metrics to compare their efficiency. They first identify the key elements for these metrics. Their results suggest relative recall and symmetric Klosgen are the most effective metrics. In addition, for multiple-fault programs, symmetric Klosgen, relative Ochiai, relative F1, and enhanced Tarantula all performed similarly well.

Feyzi and Parsa [252] proposed to incorporate fault-proneness analysis into statistical fault localization in order to address the fact that statistical fault localization techniques are biased by data collection from distinct executions of a program under analysis. Their evaluation shows that such combination is beneficial to fault localization.

1.3.4 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 [253], 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 [192] and Zeller and Hildebrandt [254] propose a technique, delta debugging, by contrasting program states between executions of a successful test and a failed test via their memory graphs described in [255]. 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 [192, 254] is orthogonal to other techniques, as it significantly reduces the original execution length. The delta tool [256] has been widely used in industry for automated debugging. In [193], 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 [257–259] 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. [260] 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 [261–263]. 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 [264] propose a cause inference model, *comparative causality*, to provide a systematic technique explaining the difference between a failed execution and a successful execution. Hashimoto et al. [265] propose a rule-based approach to minimize the set of changes to facilitate delta debugging. It uses tree differencing on ASTs to decompose changes into independent components both syntactically and semantically so that invalid subsets that do not result in testable programs can be avoided.

Predicate switching [266], 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 that, 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 [267] 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 [268] 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. Li et al. [269] propose minimum debugging frontier set (MDFS) to reduce the state exploration cost. Given an observed and reproducible failure, its execution trace is analyzed and successively narrowed by cutting the dynamic dependence graph into

two parts from the corresponding trace points. Based on the result of sparse symbolic exploration, one part is removed from further exploration. This process continues until the fault is reached. The set of statement instances in the chosen cut is called a MDFS.

Jeffrey et al. [270] 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 [271], 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.

1.3.5 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 (e.g. [272–276]).

Wong and Qi [277] propose a fault localization technique based on a back-propagation (BP) neural network, one of the most popular neural network models in practice [278]. 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 [279]. 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. [98] extend the BP-based technique [277] to Object-Oriented programs. As BP neural networks are known to suffer from issues such as paralysis and local minima, Wong et al. [201] propose another approach based on radial basis function (RBF) networks, which are less susceptible to these problems and have a faster learning rate [280, 281]. 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: (i) a method for representing test cases, coverage information, and execution results within a modified RBF neural network formalism, (ii) an innovative algorithm to simultaneously estimate the number of hidden neurons and their receptive field centers, and (iii) a weighted bit-comparison-based distance (instead of the Euclidean distance) to measure the distance between the coverage of two test cases.

In [282], 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 [107] to form a ranking. These individual rankings are then consolidated to form a final statement ranking that can be examined to locate the faults. Jonsson et al. [283] build a supervised linear Bayesian model based on the text information extracted from historical bug reports to predict where bugs are located in a component.

Mariani et al. [284] present LOUD for localizing faults in cloud systems. It first uses machine learning to detect anomalies in KPIs and reveal causal relationships among them. Later, it employs graph centrality algorithms to localize the faulty resources responsible for generating and propagating anomalies. Kim et al. [285] propose PRINCE that uses genetic programming to train a statement suspiciousness ranking model using information extracted from program spectrums of both original and mutated programs, program dependency, and program structural complexity. Zhang et al. [286] propose PRFL, which combines SBFL with the PageRank algorithm. Given the original program spectrum information, PRFL uses PageRank to recompute the spectrum information by considering the contributions of different tests. Then, SBFL techniques can be applied on the recomputed spectrum information. The combination of fault localization and defect prediction techniques can further improve the accuracy of both processes [287]. Sohn and Yoo [288] extend SBFL with code and change metrics such as size,

age, and code churn that are used in defect prediction. Using suspiciousness values from existing SBFL formulas and the source code metrics as features, Genetic Programming and linear rank Support Vector Machines are applied to learn these features for fault localization. Li et al. [289] propose DeepFL, which uses TensorFlow to learn the spectrum-, mutation-, complexity-, and textual-based features for fault localization.

1.3.6 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 [290, 291]. 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 [292].

Nessa et al. [293] 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. [294, 295] 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 [296], 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. [297] propose a technique that reinterprets 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.

Bian et al. [298] present EAntMiner, which applies a divide-and-conquer approach to exclude irrelevant statements that are irrelevant to certain critical operations and transform representations of the same logic into a canonical form. Later a *k*-nearest neighbors (*k*NN)-based method is developed to identify bugs that are difficult to be detected due to the interferences of return statements that are form identical but semantics different.

Hanam et al. [299] propose BugAID, a data mining technique for discovering common unknown bug patterns. BugAID uses unsupervised machine learning to identify language construct-based changes distilled from AST differencing of bug fixes in the code.

1.3.7 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 [300], 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 behavior of a model and the actual observed behavior of the program are used to help find bugs in the program [301, 302]. On the other hand, for model-based software fault localization [155, 303–315], 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 misbehavior. As demonstrated by the Java diagnosis experiments (JADE) in [316, 317], model-based software fault localization can be viewed as an application of model-based diagnosis [318].

Dependency-based models are derived from dependencies between statements in a program, by means of either static or dynamic analysis. Mateis et al. [308] 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 behavior of the target program. This dependency-based model is then extended to handle unstructured control flows in Java programs [319, 320], such as exceptions, recursive method calls, and return and jump statements. The notion of a dependence graph has also been extended to model behavior of a program over a test suite. Baah et al. [304] use a probabilistic PDG to model the internal behavior of a program, facilitating probabilistic analysis and reasoning about uncertain program behavior, especially those that are likely associated with faults.

Xu et al. [321] propose to do fault localization based on a single failed execution. They consider debugging as a probabilistic inference problem where the likelihood of each executed statement/variable being correct/faulty is represented by a random variable. Human knowledge, human-like reasoning rules, and program semantics are modeled as probabilistic constraints, which can be solved to identify the most likely faulty statements. Yu et al. propose to reason about observed program failures using a Bayesian Network based on probabilistic PDG s to pinpoint suspicious code entities [322, 323].

Wotawa et al. [313] use first-order logic to construct dependency-based models based on source code analysis of target programs to represent program structures and behavior. 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 [155]) 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 [310] propose an abstraction-based model in which abstract interpretation [324, 325] 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 [326].

In addition to dependency-based and abstraction-based models, value-based models [327, 328] 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 [302].

We now discuss model checking-based fault localization techniques that rely on the use of model checkers to locate bugs [190, 329–336]. 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 [15].

Ball et al. [329] 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 [333], 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, for a given counter-example, Groce and Visser use a technique to generate additional execution paths such that they are *close* to the path of the counter-example but different in a small number of actions. A metric [15, 190] based on the theory of *causality* and *counterfactual reasoning* [90, 189] is proposed to measure the distance between two execution paths. A tool, *explain* [334], is implemented to support their technique. Additional execution paths so generated may or may not cause a failure. Model components in the failed paths but not in the successful paths are possible bug locations. Chaki et al. further extend Groce's technique by combining it with *predicate abstraction* [330]. However, these techniques [15, 190, 329, 330, 333, 334] all require at least one successful execution.

Griesmayer et al. [331, 332] 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 model so that the counter-example will not fail in the revised model. Assuming there is only one bug in one model component, Griesmayer et al. propose a technique with two steps: (i) revising the program specification in such a way that if any one component in the original model is changed, then the original specification cannot be satisfied and (ii) creating variants of the original model such that each variant has exactly one component replaced by a different component with an *alternative behavior*. For each model 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 [331] use the model Checker CBMC, whereas extended studies using an additional model checker SATABS are reported in [332].

Based on a similar idea described in [331, 332], Könighofer and Bloem [335] use symbolic execution to locate bugs for imperative programs. An important point stated by Griesmayer [332] is that the extensive use of a model checker makes their techniques less efficient (in terms of time) than those in [190, 329, 330, 333, 334]; 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 model so that test cases that fail on the original model can be executed successfully on the modified model [331, 332, 335] is also used in other studies for automatic bug fixing [337–340].

Additional model-based fault localization techniques also exist. They can be applied to functional programs [341], hardware description languages like VHDL [342, 343], and spreadsheets [47, 344]. Studies such as [345, 346] make use of constraint solving, in which programs are automatically compiled into a set of constraints. Shchekotykhin et al. [312] identify the preferred system diagnosis by determining a subset of minimal conflicts. Then, a set of minimal hitting sets for this subset of conflicts can be derived to find the true causes of unexpected behavior. In [347], an expert-system approach called FLABot was proposed to assist developers in fault-localization tasks by reasoning about faults using software architecture models. Chittimalli and Shah [348] propose an approach that applies the fault localization technique to BPMN models. In the approach, the test scripts are generated per the BPMN model, and then executed by the test automation tool to produce an execution traceability matrix. The process model of test scripts now acts as source entities in the fault localization techniques, namely, Tarantula and statistical bug isolation (SBI). Finally, the analysis of the test scripts can lead the testing team to conclude root for the failure. In [305], 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 behavior 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. Lehmann and Pradel [349] propose DBDB to test real-life debuggers. It builds a finite-state model to capture common features of debuggers and compares the behavior of two debuggers with generated sequences of debugging actions. The diverging behavior and other noteworthy differences indicate a potential bug in the debugger. Wang et al. [350] apply topic model to learn and rank bug patterns for bugs located in complex program loops. Troya et al. [351] use model transformations (MTs) and assertions for fault localization. Given a set of MTs (mechanisms that manipulate and transform models), a set of assertions and source models, the violated assertions will be identified. Together with the MT coverage information, the transformation rules that deal with the construction of part of the target model will be ranked according to their suspiciousness of containing a bug.

1.3.8 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. [352] 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 an impact on faulty executions of the system. In this way, root cause changes can be semiautomatically located.

In spite of using garbage collection, Java programs may still suffer from memory leaks due to unwanted references. Chen and Chen [353] 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 [354] to predict possible locations of faults using fault locations cited by similar historical bug reports retrieved from bug report managing systems (BRMS).

In [355], 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. [356] propose an interactive fault localization technique, TALK, which incorporates programmers' feedback into SBFL techniques. Each time a programmer inspects a suspicious program element in the ranking generated by a fault localization technique, they can judge the correctness of the element and provide this information as feedback to reorder 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. Lin et al. [357] propose a feedback-based fault localization technique. Given a faulty program, the execution trace is first recorded and later developers will provide light-weight feedback on trace steps. Based on the feedbacks, suspicious steps on the trace are identified. In addition, the proposed method is able to learn and approximate bug-free paths to reduce the volume of feedbacks for the debugging process. Li et al. [358] propose another interactive, feedback-based fault localization technique. They ask developers contextualized questions in terms of queries regarding the inputs and outputs related to concrete instances of suspicious method invocations.

To better understand a program's behavior, software developers must translate their questions into code-related queries, speculating about the causes of faults. Whyline [359] 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.

Cheng et al. [360] 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. [361] 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.

Liu et al. [362, 363] propose SimFL, a fault localization approach for Simulink models by combining statistical debugging and dynamic model slicing. For Simulink models, engineers not only identify whether a test case passes or fails but also routinely and explicitly determine which specific outputs are correct and which ones are incorrect for each given test case. Relying on this observation, they use a dynamic slicing technique in conjunction with statistical debugging to generate one spectrum per output and each test case. Hence, a set of spectra that is significantly larger than the size of the test suite is obtained. The authors then use this

set of spectra to rank model blocks using statistical ranking formulas (Tarantula, Ochiai, and D*).

Wang et al. [364] investigate the factors that will affect the effectiveness of IR-based fault localization on four open-source programs. They found that the quality of bug reports determines the result of IR-based fault localization. Specifically, quality reports that contain program entity names tend to result in good ranked list of suspiciousness. In addition, the authors report that in practice the ranked lists generated by the IR-based fault localization techniques do not always help users debug; they only help when the techniques can generate perfect lists for bugs without rich, identifiable information in the reports. Moreover, if the generated list is not good enough, it can even harm developers' performance by leading them to focus on the wrong files.

Recently, some studies [365–374] have applied IR 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 SBFL techniques, IR-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. Rahman and Roy [375] combine context-aware query reformulation and IR to localize faulty entities from project source. It first determines whether there are excessive program entities in a bug report, then applies reformations to the query, and finally uses the improved query for fault localization with IR. Amar and Rigby [376] argue that faulty statements should appear only in failed test logs, but not successful test logs. In light of this, they remove from failed test logs all the statements that occur also in successful test logs, and then apply IR and kNN to flag the most suspicious lines for further investigation. Le et al. [377] propose using four types of features (i.e. suspiciousness score features, text features, topic model features, and metadata features) extracted from a bug report and an FL ranking list to build a model to predict the effectiveness of an IR-based FL technique. Hoang et al. [378] propose NetML, which utilizes multimodal information from both bug reports and program spectra for bug localization. Specifically, NetML applies network Lasso regularization to cluster both bug reports and program methods based on the similarity of their suspiciousness features. This clustering enforcement allows similar bug reports or methods to reach a consensus that leads to the same bug.

Algorithmic debugging (also called declarative debugging), first discussed in Shapiro's dissertation [8] with more details in [379, 380], 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 [212, 381–386] 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 [383, 384] 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 [382], 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 [382–384] 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. Christ et al. [381] address this problem by reporting an extended study based on error invariants [382] 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. Lamraoui and Nakajima [385] propose a formula-based fault localization method for automatic fault localization, which combines the SAT-based formal verification techniques with Reiter's model-based diagnosis theory. They implement their method by following the MaxSAT approach and the using Yices SMT solver. Their method gives a high performance in both single and multiple fault problems according to experiments using their tool SNIPER. Le et al. [212] propose Savant, a new fault localization approach that employs a learning-to-rank strategy, using likely invariant diffs and suspiciousness scores as features, to rank methods based on their likelihood of being a root cause of a failure. Savant has four steps: method clustering and test case selection, invariant mining, feature extraction, and method ranking. Savant then produces a ranked list of potentially buggy methods. However, such learning-to-rank strategy can be greatly influenced by the size of the recommended files with respect to the efficiency in detecting bugs [387]. Roychoudhury and Chandra [386] give a discussion about computer-assisted Debugging techniques. First, the authors present a major challenge in debugging – the lack of specifications capturing the intended behavior of the program. Then, they discuss how the symbolic execution techniques help debugging against this challenge. Finally, they give a forward-looking view of symbolic analysis used for automated program repair.

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 [16] 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. Wu et al. [388] conduct an empirical study to characterize the crash inducing changes. Later, a learning model that uses these features and historical crash data is built to locate crashing changes from a given set of crash reports.

Chen et al. [389] apply SFBL to diagnose problems in SDN network. The SDN-based coverage matrix consists of entries of flow rules, test cases constructed by rows and columns of association table, as well as the result vector that represents the state of the last judged network behavior.

Christi et al. [390] apply SFBL to reduce search space in test-based software modification (TBSM) when building resource-adaptive software. The modification is the diff between the original program and the adaptation. The purpose of resource adaptations is to avoid faults in correctness or performance that occur in low-resource settings. In order to map FL to TBSM, the "faulty code" in TBSM is the code that needs to be modified or removed for an adaptation, the "failing tests" are the labeled tests that are marked as pertaining to a feature to be removed from the program, and the "passing tests" are the unlabeled tests or retained tests.

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 [247], the authors propose an algorithm that iteratively samples a lower dimensional projection of the program space and identifies candidate-relevant points. Refer to Section 1.7.7 for more discussions.

1.3.9 Distribution of Papers in Our Repository

Figure 1.3 shows the distribution of papers in our repository across all categories. Spectrum-based is the most dominant category with 34% of all the papers[3] followed by model-based, which contains 17%, and sliced-based, which contains 16%. The number of papers in each of the statistics-based, program state-based, spreadsheet-based, and others categories is between 5 and 9%, followed by machine learning-based, which is 3%. The data mining and IR-based categories have the fewest number of papers, constituting only 2% each.

We present below the distribution using a different classification: static and dynamic slice-based, execution slice and program spectrum-based, and other techniques (see Endnote 3 for the rationale). Figure 1.4 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

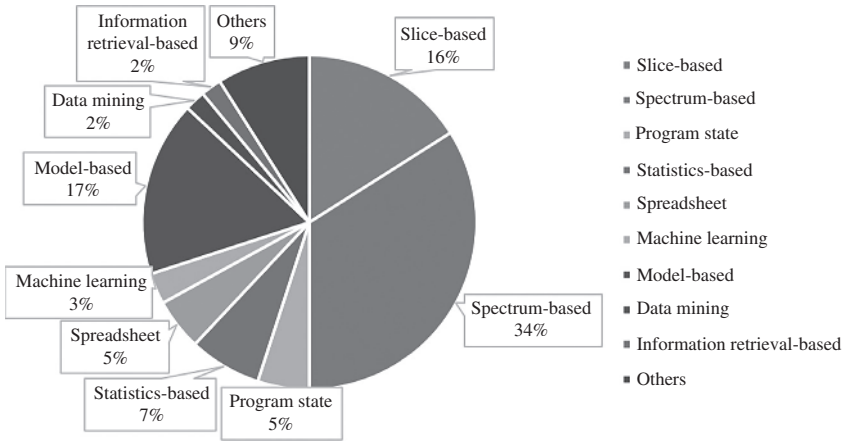

Figure 1.3 Distribution of papers in our repository.

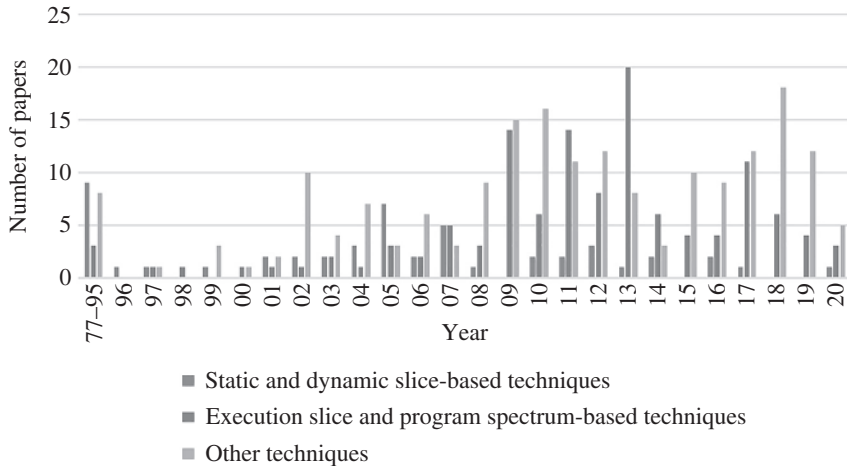

Figure 1.4 Number of papers published each year with respect to three different categories.

counts papers from 2016, and those in between give the number in the corresponding year. Figure 1.5 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.

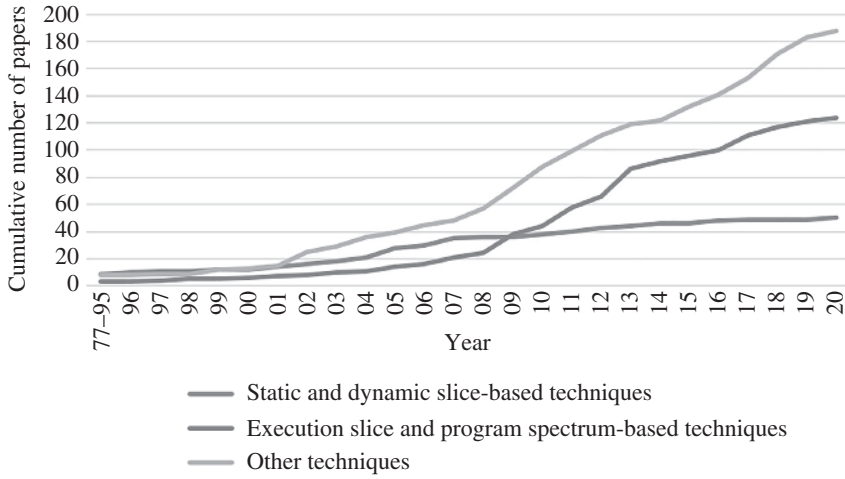

Figure 1.5 Cumulative number of papers with respect to three different categories.

- 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.

1.4 Subject Programs

Table 1.6 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, with less than 600 lines of code (excluding 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 [214, 391–393], 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

Table 1.6 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	106
Siemens: schedule	412	Priority scheduler	C	102
Siemens: print_tokens	565	Lexical analyzer	C	102
Siemens: replace	563	Pattern recognition	C	100
Siemens: print_tokens2	510	Lexical analyzer	C	98
Siemens: schedule2	307	Priority scheduler	C	98
Siemens: tot_info	406	Information measure	C	97
grep	12 653	Command-line utility for searching plain-text data sets	C	38
space	9126	ADL Interpreter	C	36
gzip	6573	Data compression	C	36
sed	12 062	GNU batch stream editor	C	20
flex	13 892	Lexical analyzer generator	C	18
NanoXML	7646	XML parser	Java	17
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

Table 1.6 (Continued)

Name	Size (Lines of code)	Brief description	Language	Number of papers
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	2700	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	4263	A PHP/MySQL solution for administering schools	PHP	4
FAQforge	734	A tool for creating and managing documents	PHP	4
webchess	2226	An online chess game	JS and PHP	4
jtopas	5400	Text parser	Java	4
timeclock	13 879	A web-based clock system	C	3
phpsysinfo	7745	Displays system information, e.g. uptime, CPU, and memory	C	3
TCC	1900	A small and fast compiler for the C programming language	C	3
Xerces	52 528	XML parser	C++	3
Mozilla Firefox	21M	Web browser	C and C++	3
tidy	31 132	A text editor for editing web content	C++	3

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.

1.5 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 percentage of code[4] that needs to be examined before the first faulty location for a given bug is identified.

The T-score [188, 214] estimates the percentage of code a programmer need not examine before the first faulty location is found. A PDG 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:

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

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 [193]. Furthermore, it assumes that programmers can follow the control- and/or data-dependency relations among statements while searching for faults.

The EXAM [117, 144, 201, 202, 277] or *Expense* [107] score is the percentage of statements in a program that has to be examined until the first faulty statement is reached:

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

In [107], the authors use the executable statements instead of the total number of statements. For techniques such as [394] 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 [123] defined as follows uses the same approach:

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

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 [117, 197, 201, 202, 277] 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 [105]:

$$\text{N-score} = \frac{N_{\text{detected}}}{N_{\text{statistic}}} \times 100\%$$

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 [197, 203]. 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 [117, 197, 201, 202]. 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 [139, 395]) or only the best case (such as the P-score in [123]) 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 [117, 201, 202, 277] 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* [188] 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 [396] indicates that such an assumption does not always hold in practice. If so, the number of statements that need to be examined to find the bug may increase. Xie et al. [397] report that fault localization techniques might even slightly weaken programmers' abilities in identifying the root faults. On the other hand, Xia et al. [398] suggest that the studies [396] and [397] suffer from several drawbacks: (i) only using small-sized programs, (ii) only involving students, and (iii) only using dated fault localization techniques. The discussion by Le et al. [399] is also similar. In response, Xia et al. conduct a study based on four large-sized open-source projects with professional software programmers. They find that fault localization techniques can help professionals reduce their debugging time, and the improvements are statistically significant and substantial. To investigate how fault localization should be improved to better benefit practitioners, Kochhar et al. [204] highlight some directions by conducting a literature review.

There are other factors that may affect the effectiveness of a software fault localization technique. Bo et al. [100] 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 1.7.2. Monperrus [400] 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 [76]), 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 [401, 402].

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

[396]. 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.

1.6 Software Fault Localization Tools

One challenge for many empirical studies on software fault localization is that they require appropriate tool support for automatic or semiautomatic data collection and suspiciousness computation. Table 1.7 gives a list of commonly used tools, including name, a brief description, availability, and which papers use the tool. Out of the 82 tools, three are commercial, 20 are open source, 12 are openly accessible but the source code is not available, and the rest may be acquired by contacting their authors.

Table 1.7 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	[95]
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	[403]
Atomizer	A dynamic atomicity checker	Via author	[404]
AUTOFLOX	Automated fault localization tool based on dynamic backward slicing	Via author	[150]
ATAC/χslice	Slicing and dicing tool for ANSI C programs	Via author	[140, 166, 175]
BARINEL	Framework to combine spectrum-based fault localization and model-based diagnosis	Via author	[318]
BigDebug	Interactive debugger for big data analytics in Apache Spark	Via author	[405, 406]

(*Continued*)

Table 1.7 (Continued)

Name	Brief description	Availability	Papers using the tool
BigSift	Debugging tool kit for big data analytics	Via author	[405, 407]
BugAssist	Fault localization tool for ANSI-C programs	Via author	[383, 384]
BugFix	A machine learning-based tool for program debugging	Via author	[408]
C2V	Validation tool for C code coverage tools	Via author	[409]
Chianti	Impact analyzer of program changes for Java programs	Via author	[26]
Chislice	Execution slicing tool	Via author	[166]
Chord	Debugging tool for concurrent program	Via author	[410]
CIL framework	Tool for extracting control flow graph and data flow information from C programs	Via author	[304]
Clover	Tool for collecting execution trace information for Java programs	Commercial	[101]
CnC	Static checking and testing tool	Openly accessible	[411]
CPTEST	A framework for automatic fault detection, localization, and correction of constraint programs	Via author	[412]
CRADLE	Validation tool for deep learning libraries	Via author	[413]
Crisp	Eclipse plug-in for constructing intermediate versions of a Java program that is being edited	Via author	[26]
Daikon	Dynamic invariant detector	Open source	[207, 414]
DBGBENCH	Tool benchmark for debugging in practice	Openly accessible	[415]
DejaVu	Regression test selection tool	Via author	[181]
Delta	Tool for delta debugging	Open source	[256]
Diablo	A link-time optimizer	Open source	[260]
DiffJ	Tool for comparing different versions of programs to find bugs	Open source	[139]

Table 1.7 (Continued)

Name	Brief description	Availability	Papers using the tool
Doxygen	Source code documentation generator and static analysis	Open source	[352]
DrDebug	Debugging tool integrating dynamic slicing and GDB debugger	Open source	[142, 154]
ESC/Java	Compile-time program checker to detect precondition violations	Open source	[411]
FindLeaks	Aspect-based tool to locate memory leaks in Java programs	Via author	[353]
Gcov	Profiling tool to collect program spectra	Open source	[100, 104, 223, 416]
GNU GDB	A debugger developed by GNU	Open source	[83]
gprof	GNU's profiling tool	Open source	[88]
GoalDebug	Constraint-based spreadsheet debugging tool	Via author	[417]
GZoltar	An automated testing and debugging framework	Openly accessible	[418, 419]
HOLMES	Statistical debugging tool	Via author	[240]
HSFal	Hybrid slice spectrum fault locator	Via author	[144]
iFL	A support tool for interactive fault localization in Eclipse IDE	Open source	[420]
JaCoCo	Java code coverage library	Open source	[419]
Jaguar	A spectrum-based fault localization tool	Open source	[421]
JARDIS	Debugging tool for JavaScript/Node.js	Via author	[422]
JavaPDG	A new platform for program dependence analysis	Openly accessible	[423]
JCoverage	A tool for coverage analysis	Open source	[215]
JCrasher	Java test cases generator to exhibit the error	Open source	[411]
Jhawk	Java static analysis tool	Commercial	[97]
JMutator	Mutation tool using seven mutation operators for Java programs	Open source	[424]
JTracor	Tool for collecting execution trace for Java programs	Via author	[424]

(*Continued*)

Table 1.7 (Continued)

Name	Brief description	Availability	Papers using the tool
JUMBLE	Tool for detecting destructive races	Via author	[425]
Phoenix Framework	A framework for developing compilers as well as program analysis, testing, and optimization	Openly accessible (from Microsoft)	[240]
MCFuzz	Debugging tool for software model checkers	Via author	[426]
MDebugger	A model-level debugger for RTE systems in the context of UML-RT	Via author	[427]
Microsoft Visual Studio Debugger	A debugging tool embedded in Microsoft Visual Studio	Commercial	[84]
MZoltar	Automatic debugging tool for android applications	Via author	[419]
N-Prog	Tool for bug detection and test case generation using random mutation and N-variant systems	Via author	[428]
NonDex	Tool for detecting and debugging wrong assumptions on Java API specifications	Via author	[429]
Pinpoint	Fault localization tool using Jaccard coefficient	Via author	[95]
Penelope	Tool for atomicity violations detection	Via author	[430]
RADAR	Debugging tool for regression problems in C/C++ programs	Openly accessible	[431]
RacerX	Debugging tool for concurrent program	Via author	[432]
Reactive Inspector	Debugger for reactive programs integrated with Eclipse Scala IDE	Open source	[433]
RxFiddle	Visualization and debugging tool for reactive extensions	Via author	[434]
Signpost	Tool for matching program behavior against known faults	via author	[435]
SLAM toolkit	Debugging tool using static analysis	Openly accessible	[329]
SLforge	CPS tool chain testing scheme	Via author	[436]

Table 1.7 (Continued)

Name	Brief description	Availability	Papers using the tool
SLOCCount	Tool for counting executable statements	Open source	[214, 241]
SmartDebug	Interactive debug assistant for Java	Via author	[437]
Spyder	Back-tracing debugger based on dynamic slicing	Via author	[140]
SNIPER	A formula-based fault localization tool	Via author	[385]
Tarantula	Fault localization tool using Tarantula	Openly accessible	[95, 175, 214]
TPTP	Eclipse plugin for profiling	Openly accessible	[89]
VART	Eclipse plugin for debugging regression faults	Openly accessible	[438]
VIDA	Visual interactive debugging tool	Via author	[439]
VHDLDIAG	A VHDL fault localization tool based on model-based diagnosis	Via author	[343]
WhoseFault	Debugging assignment tool	Via author	[440]
Whyline	An interactive debugging tool	Openly accessible	[359]
Xlab	X window system events recorder	Open source	[361]
Zoltar	Spectrum-based fault localization tool	Via author	[199]
Zoltar-M	Tool for detecting multiple bugs	Via author	[416]
χProf	Tool using execution trace to locate performance bottlenecks	Via author	[171]
χRegress	Regression test set minimization tool using program coverage and execution cost	Via author	[171]
χSuds	Tool for collecting execution trace information for C programs	Via author	[117, 171, 173, 178, 197, 201–203, 277]
χVue	Heuristics involving the control graph, execution trace, and the maintainer's knowledge to help locate features and identify feature interactions	Via author	[171]

1.7 Critical Aspects

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

1.7.1 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) [441]. However, this is not the case for real-life software, which in general contains multiple bugs [442]. Results of a study [443] 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 [444] 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 [104, 270, 394, 401, 445–457].

A popular assumption is that multiple bugs in the same program perform independently [458]. Debroy and Wong [401] 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 [104] 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. Perez et al. [441] study the prevalence of single-fault fixes in open-source Java projects and suggest that a software application may have many dormant bugs; however, they can be detected and fixed individually, therefore constituting single-faulted events. 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 [197] and a reasoning fault

localization technique based on a Bayesian reasoning framework [318]. A potential weakness of most techniques based on Bayesian reasoning (e.g. [318, 459, 460]) is that they all assume program components fail independently; in other words, interferences among multiple bugs are ignored, which is not necessarily the case in practice.

In [449], 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 [107], 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. [450] 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 [452] use the *Weil–Kettler algorithm*, a technique widely used in integer linear programming, to cluster failed test cases. It is very critical to choose the right clustering algorithm, as misgrouping can significantly decrease the FL effectiveness for programs with multiple bugs [461].

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 [462, 463] further investigate the *due-to* relationship between failed tests and underlying bugs. They apply SOBER [443] 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 [202], RBF [201], and DStar [197]) instead of Tarantula or SOBER, or using only a subset, rather than all, of the successful tests (see Section 1.7.2). These variations are yet to be explored.

Gao and Wong [464] propose MSeer for locating multiple bugs in parallel. It first uses a revised Kendall tau distance to measure the distance between two failed tests, and then applies an approach to simultaneously estimate the number of clusters and assign initial medoids to these clusters. Later, an improved K-medoids

clustering algorithm is implemented to identify failed tests and their corresponding bugs. Their case studies suggest that MSeer outperforms Jones's technique [449] in terms of both effectiveness and efficiency for parallel debugging.

1.7.2 Inputs, Outputs, and Impact of Test Cases

In addition to failed and successful test cases, many (although not all) techniques discussed in Section 1.3 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 and predicate).

The output of many spectrum-based (Section 1.3.2) 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 1.3). 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 1.3.1) only return a set of statements without specific ranking. Referring to Table 1.2, 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 Section 1.3.3 (statistics-based), Section 1.3.5 (machine learning-based), and Section 1.3.6 (data mining-based) are likely[5] to generate outputs in terms of suspiciousness rankings similar to those generated by the spectrum-based techniques, whereas program state-based (Section 1.3.4) and model-based (Section 1.3.7) 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 [93, 465, 466]. Some fault localization techniques (e.g. [166, 188, 192, 193, 221, 331, 332]) 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. [107, 117, 176, 178, 201, 202, 213, 214, 277]) use multiple failed and successful test cases. These latter techniques take advantage of more test cases than the former, so it is likely that the latter are more effective in locating program bugs. For example, Tarantula [107], which uses multiple failed and multiple successful tests, has been shown to be more effective than

nearest neighbor [188], 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 [193] 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 [467, 468] 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 [469–472]. 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 (successes or failures) 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. [473] use a meta-heuristic technique (a so-called Tabu Search approach) to automatically generate a test suite to obtain maximum branch coverage. In [403, 474, 475], 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 predefined threshold of statements coverage is reached, a sufficient number of faults are detected, or the time budget is exhausted. Xu et al. [476] introduce a bug detection mechanism for Python programs. The mechanism first collects an execution trace, and later encodes this trace and branches that are unexecuted to symbolic constraints. By solving these constraints, potential bugs as well as their triggering inputs can be identified. Jiang et al. [477] suggest that test suites satisfying branch coverage are better than those satisfying statement coverage in effectively supporting fault localization, whereas Jiang et al. [478] claim that test suites satisfying MC/DC

coverage are better than those satisfying branch coverage. Furthermore, in [479], 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 [100, 477, 480–496]. 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 [497]. In [498, 499], 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. Röβler et al. [500] 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 [191] applies a similar test case generation approach, but the generated test cases are instead used with a SBFL technique to rank basic blocks in descending order according to their suspiciousness values. Hui [489] propose a test case generation technique GIA for fault localization, which combines genetic algorithm and artificial immune algorithm. Kim et al. [491] propose the Sungkyunkwan enhanced method, which is a fault localization method with a test case optimization technique. Zhang et al. [496] propose and evaluate the strategy to remove redundant test cases with repeated spectrum in coverage information. Results show that test cases are reduced by 58–99% on average without losing the performance of fault localization. Li et al. [492] propose a test case selection strategy using the concept of dynamic basic block (DBB) to select test cases that can potentially distinguish non-faulty statements from faulty statements. First, they identify all the DBBs such that the statements in the same DBB are covered by the same test cases in the target program. Then, they identify all the groups in the target program such that any two DBBs in the same group are covered by the same test cases. Finally, they identify one failed test case and use it to initiate the test case selection procedure. Li et al. [501] apply genetic algorithm to generate test cases for software product lines with the integration of FL techniques. Each test case is first converted into a unique binary string. Then, existing test cases are modified via the use of evolutionary operators such as crossover and mutation to create new test cases. This process is repeated until a predefined coverage criterion is satisfied. Then, the test set

created to test a product is evaluated by FL techniques and reused to test another product of the same family. Liu et al. [502] propose two test case selection strategies to assist debugging process in real life. The first uses coefficient of variance (CV) to reveal failed test cases. The higher the CV of suspiciousness score of a failed test case, the more complex will be the distribution of suspiciousness scores of the statements covered by this test case, therefore requiring more effort for failure comprehension. The second strategy identifies coincidentally correct test cases and compares them with similar successful and failed test cases for better fault diagnosis. Chen et al. [503] suggest clustering the failed and successful test cases into different groups, each group having the same execution path. They improve FL effectiveness by selecting one failed group and its nearest successful group. Lekivetz and Morgan [504] argue that prior knowledge can help assume potential input combinations and finally identify failure-causing input combinations by analyzing their occurrence frequency in a test suite.

Baudry et al. [424] 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. Perez et al. [505] evaluate a test suite's diagnosability for test optimization according to its density (how frequent the components are involved with tests), diversity (to what extent the combinations of components are distributed throughout the input domain), and uniqueness (to what extent the spectra related to the components are distinguishable). Studies such as [111, 486] focus on a cross-evaluation of the impacts of different test reduction and prioritization techniques on the efficiency of software fault localization.

Based on manual analysis of more than 100 bug reports and triggering tests, Just et al. [506] find that developer-provided tests supply more information for fault localization than user-provided tests. On the other hand, developer-provided tests may overestimate a technique's ability to rank a statement by suspiciousness.

Test execution sequence also has an impact on program debugging [507]. 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.

1.7.3 Coincidental Correctness

The concept of *coincidental correctness*, introduced by Budd and Angluin in [508], 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 [329, 509–517].

Coincidental correctness can negatively impact the effectiveness of fault localization techniques. Ball et al. [329] claim that this is the reason why their technique fails to locate bugs in 3 out of 15 single-bug programs. Wang et al. [514] 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 [512] 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 [518]. Liu et al. [517] propose to deal with coincidental correctness using a weighted fuzzy classification approach to identify and manipulate coincidentally correct test cases for fault localization. Bandyopadhyay and Ghosh [509] 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. [515] 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. Zhang et al. [519] 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. Zhou et al. [520] propose a new fault-localization approach based on the probability of coincidental correctness estimated via data-flow and control-flow analyses. They first estimate the probabilities of wrong temporal values of variables in memory generated by faulty statements that do not affect the final outputs, and then apply the control flows of the statements that have use-definition dependencies on these values to revise the probabilities.

1.7.4 Faults Introduced by Missing Code

One claim that can generally be made against fault localization techniques discussed in this chapter 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 1.3.2), statistics-based (Section 1.3.3), and program state-based techniques (Section 1.3.4). 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.

1.7.5 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 [101, 521], 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 et al. [522] and Tang et al. [523] independently propose normalization methods to combine results of different fault localization techniques.

In [200], Abreu et al. address the inherent limitations of SBFL techniques, stating that component semantics of the program are not considered. They propose a way to enhance the diagnostic quality of a SBFL 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 [524], Wang et al. use two different search algorithms, simulated annealing and genetic algorithm, to find approximate optimal compositions from 22 existing

SBFL techniques. However, a search-based approach lacks flexibility and efficiency [525]. 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 [139, 395, 526, 527]. For example, in [139], 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 [144], 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 [395] emphasize that SBFL techniques (e.g. Ochiai [95]) 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 [156]) sometimes produce an undesirable ranking with statements (such as constructors), which are executed by many test cases, at the top. To eliminate these drawbacks, there have been attempts to combine techniques of these two types [528, 529]. Similar in nature, the work of Christi et al. combine delta debugging with SBFL to focus the localization to the relevant parts of the program [530].

Other combinations have also been explored. Xuan and Monperrus [531] propose Multric, a learning-based approach to combining multiple fault localization techniques. In [403], 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 [474]. In [532], 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 [533], 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.

1.7.6 Ties Within Fault Localization Rankings

As discussed earlier (in Section 1.3.2), statements with the same suspiciousness are tied for the same position in a ranking. Results of a study by Xu et al. [120], 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 1.5). 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 [117, 120, 202, 224].

1.7.7 Fault Localization for Concurrency Bugs

Concurrent programs suffer most from three kinds of access anomalies: data race [534, 535], atomicity violation [536–538], and atomic-set serializability violations [138, 329].

Among the approaches that have mushroomed in recent years, predictive analysis-based techniques haven drawn significant attention [404, 537–541]. 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 6 of 97 reported atomicity violations in a study using Atomizer (a dynamic atomicity checker) are real [404]. On the contrary, a study in [430] using a different tool, Penelope, for atomicity violations detection reports no false positive.

Tools such as Chord [410] and RacerX [432] 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 [462, 535, 542]), 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 [543–546]. For instance, Shacham et al. [546] 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 [425]. Park et al. apply a CTrigger testing framework [547] 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 [548]. Wang et al. [549] propose a technique to locate buggy shared memory accesses that are responsible for triggering concurrency bugs. Torlak et al. [550] propose a tool, MEMSAT, to help in debugging memory models. Koca et al. [551] locate faults in concurrency programs using an idea similar to SBFL techniques. Xu et al. [552] apply delta debugging to identify the threads and method invocations that are essential for causing the failure, while other threads and method invocations are removed to obtain a smaller stress test for concurrent data structures. The new execution is forced to replay the original failed execution trace, and guided back to the failed trace when the execution diverges.

1.7.8 Spreadsheet Fault Localization

Spreadsheet systems represent a landmark in the history of generic software products. It is estimated that 95% of all US firms use spreadsheets for financial reporting [553], 90% of all analysts in the industry perform calculations in spreadsheets [553], and 50% of all spreadsheets are the basis for decisions [554]. Such wide usage, however, has not been accompanied by effective mechanisms for bug prevention and detection, as shown by studies such as [555, 556]. 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) website (http://www.eusprig.org). A recent study by Reinhart and Rogoff [557] also gives a similar conclusion. In response to this, many studies regarding spreadsheet fault localization have been reported [344, 558–569].

A model-based spreadsheet fault localization technique is presented in [344], using an extended hitting-set algorithm and user-specified or historical test cases and assertions to identify possible error causes. Hofer et al. [565] 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. [570, 571], taking a

spreadsheet and one test case as input to compute a set of faulty candidates. Getzner et al. [572] propose using dynamic slicing and grouping to reduce search space and using tie-breaking strategies to prioritize cells in order to further improve the effectiveness of spreadsheets debugging. Almasi et al. [573] apply a search-based approach to detect deviation failures in financial applications by generating tests to maximize the discrepancies between the newly implemented Java program and its legacy version in the form of an Excel spreadsheet. Abraham and Erwig [417] describe a tool, GoalDebug, for debugging spreadsheets, using a constraint-based approach similar to that in [565]. 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 [337] and [338]. Debroy and Wong [337] 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 [338] to generate program modifications for bug fixing.

Hofer et al. [574] evaluate the effectiveness of 42 spectrum-based techniques in spreadsheets fault localization. 803 spreadsheets of 2 subject corpora are used in the experiment. By evaluating the scores of each spectrum-based technique in best, average, and worst scenarios, the experiment results show that Jaccard, Ochiai, and Sorensen-Dice are the best performing techniques to diagnose spreadsheets with SBFL.

Hofer and Wotawa [575] investigate the impact of erroneous cell classification on the effectiveness of SBFL on spreadsheet debugging. Cases studies on 33 spreadsheets show that SBFL still computes acceptable results in the case of erroneous cell classifications: for more than 60% of the evaluated data sets, the number of cells that must be manually inspected (before the first faulty cell is found) doubles at most when one cell value is misclassified. When there are two misclassified cell values, for more than 40% of the spreadsheets, the effort doubles at most.

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 [559]. Hermans et al. [562] 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 [563].

Other techniques aimed at reducing the occurrence of errors in spreadsheets include code inspection [576], refactoring [577], and adoption of better spreadsheet design practices [578, 579].

1.7.9 Theoretical Studies

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

Briand et al. [282] 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 (a_{ef}) to the number of successful tests (a_{es}) that execute the statement. Lee et al. [37, 580] prove that Tarantula always produces a ranking identical to that of a technique where the suspiciousness function is formulated as $a_{ef}/(a_{ef} + a_{es})$. A study by Naish et al. [113] examines over 30 formulae and divides them into groups such that those in the same group are equivalent for ranking. Independently, Debroy and Wong [102] also report a similar study showing that some similarity coefficient-based fault localization techniques are equivalent to one another. Studies such as Zhang et al. [520] focus on a cross-evaluation of the impacts of different test reduction and prioritization techniques on the efficiency of software fault localization.

Xie et al. [581] perform a theoretical study on the effectiveness of some SBFL techniques. Based on the *risk values* (which is the same as *suspiciousness* discussed in this book), 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% statement coverage. Unfortunately, some of these assumptions are oversimplified 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 [197] 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 controversy is that some advanced and effective techniques (e.g. [197, 201, 214, 318]) are excluded, even though they use exactly the same input data as those included in [581]. Le et al. [194] also question the validity of [581]. They compare the effectiveness of the five *best* fault localization techniques based on the theoretical study in [581] 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 [581]. Xie et al. [582, 583] also apply their theoretical analysis framework to genetic programming-evolved formulae and show that these formulae can be used for effective fault localization. However, they make the same oversimplified assumptions as those in [581]. In addition, Ju et al. [584] provide a theoretical analysis on the efficiency of some FL formulas in debugging programs with multiple bugs based on the number of faults that are located from certain top statements of the ranking list. However, this analysis is not applicable to one-bug-at-a-time and parallel debugging, which are mainstream strategies for multi-fault localization.

There are other theoretical studies for single-bug programs. For example, Lee et al. [585] 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 [586], Lee et al. further identify a class of *optimal* fault localization techniques for locating *deterministic* bugs (similar to Bohrbugs defined in [76]) that will always cause test cases to fail whenever they are executed. In [587], the authors revisit their previously published framework for theoretically analyzing the performance of risk evaluation formulas that are used for SBFL. Specifically, they provide justification to the assumptions/concerns of their framework such as coverage criteria, omission fault, multiple faults, and inconsistence between empirical and theoretical analyses.

1.8 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.

A publication repository has been created, including 587 papers and 68 PhD and Masters' theses on software fault localization from 1977 to 2020. These techniques are classified into nine categories: slicing-based, spectrum-based, statistics-based, machine learning-based, data mining-based, IR-based, model-based, spreadsheet-based, and additional emerging techniques. 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 oversimplified and nonrealistic 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 book to provide the software engineering community with a better understanding of state-of-the-art research in software fault localization, and identify potential drawbacks and deficiencies of existing techniques, so that additional studies can be conducted to improve their practicality and robustness.

Notes

1 In this chapter, 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."
3 Papers that only use execution slice-based techniques (e.g., [166, 178]) are included in the spectrum-based category because a statement-based execution slice is the same as ESHS (see Section 1.3.2). The slice-based category contains papers only using static slicing and/or dynamic slicing.
4 The code can be represented by statements, predicates, functions, and so on
5 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 the types of outputs generated.

References

1 Munson, J.C. and Khoshgoftaar, T.M. (1992). The detection of fault-prone programs. *IEEE Transactions on Software Engineering* 18 (5): 423–433.
2 Pai, G.J. and Dugan, J.B. (2007). Empirical analysis of software fault content and fault proneness using bayesian methods. *IEEE Transactions on Software Engineering* 33 (10): 675–686.
3 Wright, C.S. and Zia, T.A. (2011). A quantitative analysis into the economics of correcting software bugs. *Proceedings of the International Conference on Computational Intelligence in Security for Information Systems*, Torremolinos, Spain (8–10 June 2011), 198–205.
4 Burke, D. All circuits are busy now: the 1990 AT&T long distance network collapse. http://users.csc.calpoly.edu/~jdalbey/SWE/Papers/att_collapse.html (accessed January 2022).
5 Wong, W.E., Debroy, V., Surampudi, A. et al. (2010). Recent catastrophic accidents: investigating how software was responsible. *Proceedings of the 4th International Conference on Secure Software Integration and Reliability*

Improvement, Singapore, Singapore (9–11 June 2010), 14–22. IEEE. https://doi.org/10.1109/SSIRI.2010.38.

6 NIST. Software errors cost U.S. economy $59.5 billion annually. http://www.abeacha.com/NIST_press_release_bugs_cost.htm (accessed January 2022).

7 Vessy, I. (1985). Expertise in debugging computer programs: a process analysis. *International Journal of Man-Machine Studies* 23 (5): 459–494.

8 Shapiro, E. (1982). Algorithmic program debugging. PhD dissertation. Yale University.

9 Agrawal, H. (1991). Towards automatic debugging of computer program. PhD dissertation. Purdue University.

10 Pan, H. (1993). Software debugging with dynamic instrumentation and test-based knowledge. PhD dissertation. Purdue University.

11 Gregory, W.B. (1994). Logic programs for consistency-based diagnosis. PhD dissertation. Carleton University.

12 Liblit, B. (2004). Cooperative bug isolation. PhD dissertation. University of California, Berkeley.

13 Peischl, B. (2004). Automated source-level debugging of synthesizable VHDL designs. PhD dissertation. Graz University of Technology.

14 He, H. and Gupta, N. (2004). Automated debugging using path-based weakest preconditions. *International Conference on Fundamental Approaches to Software Engineering*, Barcelona, Spain (29 March to 2 April 2004), 267–280. Springer.

15 Groce, A. (2005). Error explanation and fault localization with distance metrics. PhD dissertation. Carnegie Mellon University.

16 Renieris, E. (2005). A research framework for software-fault localization tools. PhD dissertation. Brown University.

17 Kob, D. (2005). Extended modeling for automated fault localization in object-oriented software. PhD dissertation. Graz University of Technology.

18 Hovemeyer, D. (2005). Simple and effective static analysis to find bugs. PhD dissertation. University of Maryland.

19 Hu, P. (2006). Automated fault localization: a statistical predicate analysis approach. PhD dissertation. The University of Hong Kong.

20 Zhang, X. (2007). Fault localization via precise dynamic slicing. PhD dissertation. University of Arizona.

21 Vayani, R. (2007). Improving automatic software fault localization. Master thesis. Delft University of Technology.

22 Kompella, R.R. (2007). Fault localization in backbone networks. PhD dissertation. University of California.

23 Griesmayer, A. (2007). Debugging software: from verification to repair. PhD dissertation. Graz University of Technology.

24 Wang, T. (2007). Post-mortem dynamic analysis for software debugging. PhD dissertation. National University of Singapore.

25 Tallam, S. (2007). Fault location and avoidance in long-running multithreaded applications. PhD dissertation. University of Arizona.

26 Chesley, O.C., Ren, X., Ryder, B.G., and Tip, F. (2007). Crisp – a fault localization tool for java programs. *29th International Conference on Software Engineering (ICSE '07)*, Minneapolis, MN, USA (20–26 May 2007), 775–779. IEEE.

27 Lu, S. (2008). Understanding, detecting, and exposing concurrency bugs. PhD dissertation. University of Illinois at Urbana-Champaign.

28 Peischl, B., Riaz, N., and Wotawa, F. (2008). Advances in automated source-level debugging of verilog designs. In: *New Challenges in Applied Intelligence Technologies* (ed. N.T. Nguyen and R. Katarzyniak), 363–372. Berlin, Heidelberg: Springer.

29 Jones, J.A. (2008). Semi-Automatic Fault Localization. PhD dissertation, Georgia Institute of Technology.

30 Zhang, Z. (2009). Software debugging through dynamic analysis of program structures. PhD dissertation. The University of Hong Kong.

31 Abreu, R. (2009). Spectrum-based fault localization in embedded software. PhD dissertation. University of Minho geboren te Fão.

32 Jeffrey, D. (2009). Dynamic state alteration techniques for automatically locating software errors. PhD dissertation. University of California Riverside.

33 Wang, X. (2010). Automatic localization of code omission faults. PhD dissertation. The Hong Kong University of Science and Technology.

34 Pastore, F. (2010). Automatic diagnosis of software functional faults by means of inferred behavioral models. PhD dissertation. Università degli Studi di Milano Bicocca.

35 Nica, M. (2010). On the use of constraints in automated program debugging – from foundations to empirical results. PhD dissertation. Graz University of Technology.

36 Fry, Z.P. (2011). Fault localization using textual similarities. Master thesis. University of Virginia.

37 Lee, H. (2011). Spectral debugging. PhD dissertation. The University of Melbourne.

38 Debroy, V. (2011). Towards the automation of program debugging. PhD dissertation. The University of Texas at Dallas.

39 Sanchez, A. (2011). Cost optimizations in runtime testing and diagnosis. PhD dissertation. Delft University of Technology.

40 DeMott, J.D. (2012). Enhancing automated fault discovery and analysis. PhD dissertation. Michigan State University.

41 Zhang, X. (2012). Secure and efficient network fault localization. PhD dissertation. Carnegie Mellon University.

42 Xie, X. (2012). On the analysis of spectrum-based fault localization. PhD dissertation. Swinburne University of Technology.

43 Perez, A. (2012). Dynamic code coverage with progressive detail levels. PhD dissertation. University of Porto.

44 Santelices, R. (2012). Change-effects analysis for effective testing and validation of evolving software. PhD dissertation. Georgia Institute of Technology.

45 Baah, G.K. (2012). Statistical causal analysis for fault localization. PhD dissertation. Georgia Institute of Technology.

46 Sahoo, S.K. (2013). A novel invariants-based approach for automated software fault localization. PhD dissertation. University of Illinois at Urbana-Champaign.

47 Hofer, B. (2013). From fault localization of programs written in third level language to spreadsheets. PhD dissertation. Graz University of Technology.

48 Bandyopadhyay, A. (2013). Mitigating the effect of coincidental correctness in spectrum based fault localization. PhD dissertation. Colorado State University.

49 Roychowdhry, S. (2013). A mixed approach to spectrum-based fault localization using information theoretic foundations. PhD dissertation. University of Texas at Austin.

50 Ali, S. (2013). Localizing state-dependent faults using associated sequence mining. Ph.D. dissertation. The University of Western Ontario.

51 Kühnert, C. (2013). *Data-Driven Methods for Fault Localization in Process Technology 15*. KIT Scientific Publishing.

52 Qi, D. (2013). Semantic analyses to detect and localize software regression errors. PhD dissertation. National University of Singapore.

53 Sumner, W.N. (2013). Automated failure explanation through execution comparison. PhD dissertation. Purdue University.

54 Hays, M. (2014). A fault-based model of fault localization techniques. PhD dissertation. University of Kentucky.

55 Park, S. (2014). Effective fault localization techniques for concurrent software. PhD dissertation. Georgia Institute of Technology.

56 Shu, G. (2014). Statistical estimation of software reliability and failure-causing effect. PhD dissertation. Case Western Reserve University.

57 Lucia, L. (2014). Ranking-based approaches for localizing faults. PhD dissertation. Singapore Management University.

58 Moon, S. (2014). Effective software fault localization using dynamic program behaviors. Master thesis. Korea Advanced Institute of Science and Technology.

59 Liu, Y. (2015). Automated analysis of energy efficiency and performance for mobile application. PhD dissertation. The Hong Kong University of Science and Technology.

60 Chen, C. (2015). Automated fault localization for service-oriented software systems. PhD dissertation. Delft University of Technology.

61 Rohr, M. (2015). Workload-sensitive timing behavior analysis for fault localization in software systems. PhD dissertation. Kiel University.

62 Bayraktar, O. (2015). Ela: an automated statistical fault localization technique. PhD dissertation. The Middle East Technical University.

63 Tonzirul, A. (2016). Fault discovery, localization, and recovery in smartphone apps. PhD dissertation. University of California Riverside.

64 Gholamosseinghandehari, L. (2016). Fault localization based on combinatorial testing. PhD dissertation. The University of Texas at Arlington.

65 Gao, R. (2017). Advanced Fault Localization for Programs with Multiple Bugs. PhD dissertation. University of Texas at Dallas.

66 Sun, S. (2017). Statistical fault localization and causal interactions. PhD dissertation. Case Western Reserve University.

67 Wu, R. (2017). Automated techniques for diagnosing crashing bugs. PhD dissertation. The Hong Kong University of Science and Technology.

68 Roy, A. (2018). Simplifying datacenter fault detection and localization. PhD dissertation. University of California San Diego.

69 Guo, Y. (2018). Towards automatically localizing and repairing SQL faults. PhD dissertation. George Mason University.

70 Safdari, N. (2018). Learning to rank relevant files for bug reports using domain knowledge, replication and extension of a learning-to-rank approach. Master thesis. Rochester Institute of Technology.

71 Ting, D. (2019). A hybrid approach to cloud system performance bug detection, diagnosis and fix. PhD dissertation. North Carolina State University.

72 Thompson, G. (2020). Towards automated fault localization for prolog. Master thesis. North Carolina A&T State University.

73 Li, X. (2020). An integrated approach for automated software debugging via machine learning and big code mining. PhD dissertation. The University of Texas at Dallas.

74 Gulzar, M.A. (2020). Automated testing and debugging for big data analytics. PhD dissertation. University of California, Los Angeles.

75 Mathur, M. (2020). Leveraging Distributed Tracing and Container Cloning for Replay Debugging of Microservices. Master thesis. University of California, Los Angeles.

76 Grottke, M. and Trivedi, K.S. (2005). A classification of software faults. *Journal of Reliability Engineering Association of Japan* 27 (7): 425–438.

77 Avizienis, A., Laprie, J.C., Randell, B., and Landwehr, C.E. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1 (1): 11–33.

78 Edwards, J.C. (2003). Method, system, and program for logging statements to monitor execution of a program. US Patent 6,539,501 B1, filed 16 December 1999 and issued 25 March 2003.

79 Rosenblum, D.S. (1995). A practical approach to programming with assertions. *IEEE Transactions on Software Engineering* 21 (1): 19–31.

80 Rosenblum, D.S. (1992). Towards a method of programming with assertions. *Proceedings of the 14th international Conference on Software Engineering*

(ICSE '92), Melbourne, Australia (11–15 May 1992), 92–104. ACM. https://dl.acm.org/doi/pdf/10.1145/143062.143098.

81 Coutant, D.S., Meloy, S., and Ruscetta, M. (1988). DOC: a practical approach to source-level debugging of globally optimized code. *Proceedings of the ACM SIGPLAN Conference on Program Language Design and Implementation*, Atlanta, GA, USA (June 1988), 125–134.

82 Hennessy, J. (1982). Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems* 4 (3): 323–344.

83 GNU. GDB: the GNU Project Debugger. http://www.gnu.org/software/gdb/ (accessed January 2022).

84 MSDN. Debugging in Visual Studio. https://msdn.microsoft.com/en-us/library/sc65sadd.aspx (accessed January 2022).

85 Ball, T. and Larus, J.R. (1994). Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems* 16 (4): 1319–1360.

86 Hauswirth, M. and Cillmbi, T.M. (2004). Low-overhead memory leak detection using adaptive statistical profiling. *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, USA (October 2004), 156–164.

87 Runciman, C. and Wakeling, D. (1993). Heap profiling of lazy functional programs. *Journal of Functional Programming* 3 (02): 217–245.

88 GNU gprof. http://sourceware.org/binutils/docs/gprof/ (accessed January 2022).

89 Eclipse. Archived Eclipse Projects. http://www.eclipse.org/tptp (accessed January 2022).

90 Lewis, D. (1973). Causation. *Journal of Philosophy* 70 (17): 556–567.

91 Pearl, J. (2000). *Causality: Models, Reasoning, and Inference*. Cambridge University Press.

92 Abreu, R., González, A., Zoeteweij, P., and van Gemund, A.J. (2008). Automatic software fault localization using generic program invariants. *Proceedings of the ACM Symposium on Applied Computing,* Ceara, Brazil (March 2008), 712–717.

93 Abreu, R., Zoeteweij, P., Golsteijn, R., and van Gemund, A.J. (2009). A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 82 (11): 1780–1792.

94 Abreu, R., Zoeteweij, P., and van Gemund, A.J. (2006). An evaluation of similarity coefficients for software fault localization. *Proceedings of the Pacific Rim International Symposium on Dependable Computing*, Riverside, CA, USA (December 2006), 39–46.

95 Abreu, R., Zoeteweij, P., and van Gemund, A.J. (2007). On the accuracy of spectrum-based fault localization. *Proceedings of Testing: Academic and Industrial Conference Practice and Research Techniques – MUTATION*, Windsor, CT, USA (September 2007), 89–98.

96 Ali, S., Andrews, J.H., Dhandapani, T., and Wang, W. (2009). Evaluating the accuracy of fault localization techniques. *Proceedings of the 2009 IEEE/ACM*

International Conference on Automated Software Engineering, Auckland, New Zealand (November 2009), 76–87.

97 Arisholm, E., Briand, L.C., and Johannessen, E.B. (2010). A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software* 83 (1): 2–17.

98 Ascari, L.C., Araki, L.Y., Pozo, A.R.T., and Vergilio, S.R. (2009). Exploring machine learning techniques for fault localization. *Proceedings of the 10th Latin American Test Workshop,* Buzios, Brazil (March 2009), 1–6.

99 Bandyopadhyay, A. and Ghosh, S. (2011). On the effectiveness of the tarantula fault localization technique for different fault classes. *Proceedings of the IEEE International Symposium on High-Assurance Systems Engineering*, Boca Raton, FL, USA (November 2011), 317–324.

100 Bo, J., Zhang, Z., Tse, T.H., and Chen, T.Y. (2009). How well do test case prioritization techniques support statistical fault localization. *Proceedings of the Annual IEEE Computer Software and Applications Conference (COMPSAC '09)*, Seattle, WA, USA (20–24 July 2009), 99–106.

101 Debroy, V. and Wong, W.E. (2013). A consensus-based strategy to improve the quality of fault localization. *Software: Practice and Experience* 43 (8): 989–1011.

102 Debroy, V. and Wong, W.E. (2011). On the equivalence of certain fault localization techniques. *Proceedings of the ACM Symposium on Applied Computing*, TaiChung, Taiwan (March 2011), 1457–1463.

103 Debroy, V., Wong, W.E., Xu, X. and Choi, B. (2010). A grouping-based strategy to improve the effectiveness of fault localization techniques. *Proceedings of the International Conference on Quality Software*, Zhangjiajie, China (July 2010), 13–22.

104 DiGiuseppe, N. and Jones, J.A. (2011). On the influence of multiple faults on coverage-based fault localization. *Proceedings of the International Symposium on Software Testing and Analysis*, Toronto, BC, Canada (July 2011), 210–220.

105 Gong, C., Zheng, Z., Li, W., and Hao, P. (2012). Effects of class imbalance in test suites: an empirical study of spectrum-based fault localization. *Proceedings of the IEEE Annual Computer Software and Applications Conference Workshops (COMPSAC Workshops '12)*, Izmir, Turkey (July 2012), 470–475.

106 Jiang, B., Zhang, Z., Chan, W.K. et al. (2012). How well does test case prioritization integrate with statistical fault localization? *Journal of Information and Software Technology* 54 (7): 739–758.

107 Jones, J.A. and Harrold, M.J. (2005). Empirical evaluation of the tarantula automatic fault-localization technique. *Proceedings of the International Conference Automated Software Engineering*, Long Beach, CA, USA (November 2005), 273–282.

108 Kim, J. and Lee, E. (2014). Empirical evaluation of existing algorithms of spectrum based fault localization. *Proceedings of the International Conference on Information*

Networking 2014 (ICOIN '14), Phuket, Thailand (10–12 February 2014), 346–351. IEEE. https://doi.org/10.1109/ICOIN.2014.6799702.

109 Kusumoto, S., Nishimatsu, A., Nishie, K., and Inoue, K. (2002). Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering* 7 (1): 49–76.

110 Le, T.B. and Lo, D. (2013). Will fault localization work for these failures? An automated approach to predict effectiveness of fault localization tools. *Proceedings of the 29th IEEE International Conference on Software Maintenance*, Eindhoven, Netherland (September 2013), 310–319.

111 McMaster, S. and Memon, A. (2007). Fault detection probability analysis for coverage-based test suite reduction. *Proceedings of the International Conference on Software Maintenance*, Paris, France (October 2007), 335–344.

112 Naish, L., Lee, H.J., and Ramamohanarao, K. (2010). Statements versus predicate in spectral bug localization. *Proceedings of the Asia-Pacific Software Engineering Conference*, Sydney, Australia (November 2010), 375–384.

113 Naish, L., Lee, H.J., and Ramamohanarao, K. (2011). A model for spectra-based software diagnosis. *Journal of the ACM Transactions on Software Engineering and Methodology* 20 (3): 1–32.

114 Qi, Y., Mao, X., Lei, Y., and Wang, C. (2013). Using automated program repair for evaluating the effectiveness of fault localization techniques. *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, Lugano, Switzerland (July 2013), 191–201.

115 Rao, P., Zheng, Z., Chen, T.Y. et al. (2013). Impacts of test suite's class imbalance on spectrum-based fault localization techniques. *Proceedings of the 13th International Conference on Quality Software*, Nanjing, China (July 2013), 260–267.

116 Stumptner, M. and Wotawa, F. (1998). A survey of intelligent debugging. *AI Communications* 11 (1): 35–51.

117 Wong, W.E., Debroy, V., and Choi, B. (2010). A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software* 83 (2): 188–208.

118 Xu, J., Chan, W.K., Zhang, Z. et al. (2011). A dynamic fault localization technique with noise reduction for java programs. *Proceedings of the International Conference on Quality Software*, Madrid, Spain (July 2011), 11–20.

119 Xu, J., Zhang, Z., Chan, W.K. et al. (2013). A general noise-reduction framework for fault localization of java program. *Information and Software Technology* 55 (5): 880–896.

120 Xu, X., Debroy, V., Wong, W.E., and Guo, D. (2011). Ties within fault localization rankings: exposing and addressing the problem. *International Journal of Software Engineering and Knowledge Engineering* 21 (6): 803–827.

121 Zhang, X., Gupta, N., and Gupta, R. (2007). A study of effectiveness of dynamic slicing in locating real faults. *Empirical Software Engineering* 12 (2): 143–160.

122 Zhang, X., He, H., Gupta, N., and Gupta, R. (2005). Experimental evaluation of using deynamic slices for fault location. *Proceedings of the International Workshop on Automated Debugging*, Monterey, CA, USA (September 2005), 33–42.

123 Zhang, Z., Chan, W.K., Tse, T.H. et al. (2009). Is non-parametric hypothesis testing model robust for statistical fault localization? *Information and Software Technology* 51 (11): 1573–1585.

124 Zheng, J., Williams, L., Nagappan, N. et al. (2006). On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering* 32 (4): 240–253.

125 Binkley, D. and Harman, M. (2004). A survey of empirical results on program slicing. *Advances in Computers* 62: 105–178.

126 Tip, F. (1995). A survey of program slicing techniques. *Journal of Programming Languages* 3 (3): 121–189.

127 Xu, B., Qian, J., Zhang, X. et al. (2005). A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes* 30 (2): 1–36.

128 Weiser, M. (1979). Program slicing: formal, psychological, and practical investigations of an automatic program abstraction method. PhD dissertation. University of Michigan.

129 Weiser, M. (1984). Program slicing. *IEEE Transactions on Software Engineering* 10 (4): 352–357.

130 Lyle, J.R. and Weiser, M. (1987). Automatic program bug location by program slicing. *Proceedings of the International Conference on Computer and Applications*, Beijing, China (23–27 June 1987), 877–883. IEEE. ISBN: 0-8186-0780-7.

131 Liang, D. and Harrold, M.J. (2002). Equivalence analysis and its application in improving the efficiency of program slicing. *ACM Transactions on Software Engineering and Methodology* 11 (3): 347–383.

132 Kiss, Á., Jász, J., and Gyimóthy, T. (2005). Using dynamic information in the interprocedural static slicing of binary executables. *Software Quality Control* 13 (3): 227–245.

133 Tip, F. and Dinesh, T.B. (2001). A slicing-based approach for locating type errors. *ACM Transactions on Software Engineering and Methodology* 10 (1): 5–55.

134 Zhang, Y. and Santelices, R.A. (2016). Prioritized static slicing and its application to fault localization. *Journal of Systems and Software* 114: 38–53.

135 Agrawal, H. and Horgan, J.R. (1990). Dynamic program slicing. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, White Plains, NY, USA (June 1990), 246–256.

136 Korel, B. and Laski, J. (1988). Dynamic program slicing. *Information Processing Letters* 29 (3): 155–163.

137 Agrawal, H., DeMillo, R.A., and Spafford, E.H. (1993). Debugging with dynamic slicing and backtracking. *Software–Practice & Experience* 23 (6): 589–616.

138 Al-Khanjari, Z.A., Woodward, M.R., Ramadhan, H.A., and Kutti, N.S. (2005). The efficiency of critical slicing in fault localization. *Journal of Software Quality Control* 13 (2): 129–153.

139 Alves, E., Gligoric, M., Jagannath, V. and d'Amorim, M. (2011). Fault-localization using dynamic slicing and change impact analysis. *Proceedings of the IEEE International Symposium on Automated Software Engineering*, Lawrence, KS, USA (November 2011), 520–523.

140 DeMillo, R.A., Pan, H., and Spafford, E.H. (1996). Critical slicing for software fault localization. *Proceedings of the International Symposium on Software Testing and Analysis*, San Diego, CA, USA (January 1996), 121–134.

141 DiGiuseppe, N. and Jones, J.A. (2015). Fault density, fault types, and spectra-based fault localization. *Empirical Software Engineering* 20 (4): 928–967.

142 DrDebug. Deterministic replay based debugging with pin. www.drdebug.org (accessed January 2022).

143 Ishii, Y. and Kutsuna, T. (2016). Effective fault localization using dynamic slicing and an SMT solver. *Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation Workshops,* Chicago, IL, USA (April 2016), 180–188.

144 Ju, X., Jiang, S., Chen, X. et al. (2014). HSFal: effective fault localization using hybrid spectrum of full slices and execution slices. *Journal of Systems and Software* 90: 3–17.

145 Korel, B. (1988). PELAS – program error-locating assistant system. *IEEE Transactions on Software Engineering* 14 (9): 1253–1260.

146 Lian, L., Kusumoto, S., Kikuno, T. et al. (1997). A new fault localizing method for the program debugging process. *Information and Software Technology* 39 (4): 271–284.

147 Liu, C., Zhang, X., Han, J. et al. (2007). Indexing noncrashing failures: a dynamic program slicing-based approach. *Proceedings of the International Conference on Software Maintenance*, Paris, France (October 2007), 455–464.

148 Mao, X., Lei, Y., Dai, Z. et al. (2014). Slice-based statistical fault localization. *Journal of Systems and Software* 89: 51–62.

149 Mohapatra, D.P., Mall, R., and Kumar, R. (2004). An edge marking technique for dynamic slicing of object-oriented programs. *Proceedings of the International Computer Software and Applications Conference (COMPSAC '04)*, Hong Kong (September 2004), 60–65.

150 Ocariza, F.S. Jr., Li, G., Pattabiraman, K., and Mesbah, A. (2016). Automatic fault localization for client-side javascript. *Software Testing, Verification and Reliability* 26 (1): 69–88.

151 Pan, H. and Spafford, E. (1992). Heuristics for Automatic Localization of Software Faults. Technical Report, SERC-TR-116-P. Purdue University.

152 Qian, J. and Xu, B. (2008). Scenario oriented program slicing. *Proceedings of the ACM Symposium on Applied Computing,* Fortaleza, Brazil (March 2008), 748–752.

153 Sterling, C.D. and Olsson, R.A. (2005). Automated bug isolation via program chipping. *Proceedings of the International Symposium on Automated and Analysis-Driven Debugging*, Monterey, CA, USA (September 2005), 23–32.

154 Wang, Y., Patil, H., Pereira, C. et al. (2014). DrDebug: deterministic replay based cyclic debugging with dynamic slicing. *IEEE International Symposium on Code Generation and Optimization*, Orlando, FL, USA (February 2014), 98–108.

155 Wotawa, F. (2002). On the relationship between model-based debugging and program slicing. *Artificial Intelligence* 135 (1–2): 125–143.

156 Wotawa, F. (2010). Fault localization based on dynamic slicing and hitting-Set computation. *Proceedings of the International Conference on Quality Software*, Zhangjiajie, China (July 2010), 161–170.

157 Zhang, X., Gupta, N., and Gupta, R. (2007). Locating faulty code by multiple points slicing. *Software Practice and Experience* 37 (9): 935–961.

158 Treffer, A. and Uflacker, M. (2016). The slice navigator: focused debugging with interactive dynamic slicing. *Proceedings of the 2016 IEEE 27th IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW '16)*, Ottawa, ON, Canada (23–27 October 2016), 175–180. IEEE. https://doi.org/10.1109/ISSREW.2016.17.

159 Sun, C.A., Ran, Y., Zheng, C. et al. (2018). Fault localisation for WS-BPEL programs based on predicate switching and program slicing. *Journal of Systems and Software* 135: 191–204.

160 Tu, J., Xie, X., Chen, T.Y., and Xu, B. (2019). On the analysis of spectrum based fault localization using hitting sets. *Journal of Systems and Software* 147: 106–123.

161 Guo, Y., Motro, A., and Li, N. (2017). Localizing faults in SQL predicates. *Proceedings of the 2017 IEEE International Conference on Software Testing, Verification and Validation*, Tokyo, Japan (March 2017), 1–11.

162 Zhang, X., Tallam, S., Gupta, N., and Gupta, R. (2007). Towards locating execution omission errors. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, USA (June 2007), 415–424.

163 Gyimothy, T., Beszedes, A., and Forgacs, I. (1999). An efficient relevant slicing method for debugging. *Proceedings of the European Software Engineering Conference, held jointly with the ACM SIGSOFT Symposium on the Foundation of Software Engineering*, Toulouse, France (September 1999), 303–321.

164 Weeratunge, D., Zhang, X., Sumner, W.N., and Jagannathan, S. (2010). Analyzing concurrency bugs using dual Slicing. *Proceedings of the International Symposium on Software Testing and Analysis*, Trento, Italy (July 2010), 253–264.

165 Wang, X. and Liu, Y. (2015). Automated fault localization via hierarchical multiple predicate switching. *Journal of Systems and Software* 104: 69–81.

166 Agrawal, H., Horgan, J.R., London, S., and Wong, W.E. (1995). Fault localization using execution slices and dataflow tests. *Proceedings of the Sixth International Symposium on Software Reliability Engineering (ISSRE '95)*, Toulouse, France (October 1995), 143–151.

167 Beszedes, A., Gergely, T., Szabo, Z. et al. (2001). Dynamic slicing method for maintenance of large C programs. *Proceedings of the European Conference on Software Maintenance and Reengineering*, Lisbon, Portugal (March 2001), 105–113.

168 Korel, B. and Yalamanchili, S. (1994). Forward Computation of Dynamic Program Slices. *Proceedings of the International Symposium on Software Testing and Analysis*, Seattle, WA, USA (August 1994), 66–79.

169 Zhang, X., Gupta, R., and Zhang, Y. (2003). Precise dynamic slicing algorithms. *Proceedings of the International Conference on Software Engineering (ICSE '03)*, Portland, OR, USA (May 2003), 319–329.

170 Zhang, X., Gupta, R., and Zhang, Y. (2004). Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. *Proceedings of the International Conference on Software Engineering (ICSE '04)*, Edinburgh, UK (May 2004), 502–511.

171 Agrawal, H., Horgan, J.R., Li, J.J. et al. (1998). Mining system tests to aid software maintenance. *Computer* 31 (7): 64–73.

172 Bellcore (1998). *χSuds User's Manual*. Bridgewater, NJ, USA: Telcordia Technologies (formerly Bellcore). https://www.cs.purdue.edu/homes/apm/foundationsBook/Labs/coverage/xsuds.pdf.

173 Wong, W.E. and Li, J.J. (2005). An integrated solution for testing and analyzing java applications in an industrial setting. *Proceedings of the Asia-Pacific Software Engineering Conference*, Taipei, Taiwan (December 2005), 576–583.

174 Jones, J.A., Harrold, M.J., and Stasko, J. (2001). Visualization for fault localization. *Proceedings of the Workshop on Software Visualization, 23rd International Conference on Software Engineering (ICSE '01)*, Toronto, ON, Canada (May 2001), 71–75.

175 Jones, J.A., Harrold, M.J., and Stasko, J. (2002). Visualization of test information to assist fault localization. *Proceedings of the International Conference on Software Engineering (ICSE '02)*, Orlando, FL, USA (May 2002), 467–477.

176 Wong, W.E., Sugeta, T., Qi, Y., and Maldonado, J.C. (2005). Smart debugging software architectural design in SDL. *Journal of Systems and Software* 76 (1): 15–28.

177 Eric Wong, W., Gao, R., Li, Y. et al. (2016). A survey on software fault localization. *IEEE Transactions on Software Engineering* 42 (8): 707–740.

178 Wong, W.E. and Qi, Y. (2006). Effective program debugging based on execution slices and inter-block data dependency. *Journal of Systems and Software* 79 (7): 891–903.

179 Krinke, J. (2004). Slicing, chopping, and path conditions with barriers. *Software Quality Control* 12 (4): 339–360.

180 Stridharan, M., Fink, S.J., and Bodik, R. (2007). Thin Slicing. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, USA (June 2007), 112–122.

181 Harrold, M.J., Rothermel, G., Sayre, K. et al. (2000). An empirical investigation of the relationship between spectra differences and regression faults. *Journal of Software Testing, Verification and Reliability* 10 (3): 171–194.

182 Reps, T., Ball, T., Das, M., and Larus, J. (1997). The use of program profiling for software maintenance with applications to the Year 2000 problem. *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, Zurich (September 1997), 432–449.

183 Collofello, J.S. and Cousins, L. (1987). Towards automatic software fault location through decision-to-decision path analysis. *Proceedings of the International Workshop on Managing Requirements Knowledge*, Chicago, IL, USA (June 1987), 539–544.

184 Masri, W. (2015). Automated fault localization: advances and challenges. *Advances in Computers* 99: 103–156.

185 Agrawal, H., DeMillo, R.A., and Spafford, E.H. (1991). An execution backtracking approach to program debugging. *IEEE Software* 8 (3): 21–26.

186 Korel, B. and Laski, J. (1988). STAD: a system for testing and debugging: user perspective. *Proceedings of the Workshop on Software Testing, Verification, and Analysis*, Banff, AB, Canada (19–21 July 1988), 13–14. IEEE.

187 Taha, A.B., Thebaut, S.M., and Liu, S.S. (1989). An approach to software fault localization and revalidation based on incremental data flow analysis. *Proceedings of the International Conference Computer Software and Applications*, Orlando, FL, USA (20–22 September 1989), 527–534. IEEE. https://doi.org/10.1109/CMPSAC.1989.65142.

188 Renieris, M. and Reiss, S.P. (2003). Fault localization with nearest neighbor queries. *Proceedings of the International Conference on Automated Software Engineering,* Montreal, QC, Canada (October 2003), 30–39.

189 Lewis, D. (2013). *Counterfactuals*. Wiley.

190 Groce, A., Chaki, S., Kroening, D., and Strichman, O. (2006). Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer* 8 (3): 229–247.

191 Jin, W. and Orso, A. (2013). F3: fault localization for field failures. *Proceedings of the International Symposium on Software Testing and Analysis*, Lugano, Switzerland (July 2013), 213–223.

192 Zeller, A. (2002). Isolating cause-effect chains from computer programs. *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, Charleston, SC, USA (November 2002), 1–10.

193 Cleve, H. and Zeller, A. (2005). Locating causes of program failures. *Proceedings of the IEEE International Conference on Software Engineering (ICSE '05)*, Louis, MO, USA (May 2005), 342–351.

194 Le, T.B., Thung, F., and Lo, D. (2013). Theory and practice, do they match? A case with spectrum-based fault localization. *Proceedings of the IEEE International Conference on Software Maintenance*, Eindhoven, Netherland (September 2013), 380–383.

195 Choi, S., Cha, S., and Tappert, C.C. (2010). A survey of binary similarity and distance measures. *Journal of Systemics, Cybernetics and Informatics* 8 (1): 43–48.

196 Willett, P. (2003). Similarity-based approaches to virtual screening. *Biochemical Society Transactions* 31 (3): 603–606.

197 Wong, W.E., Debroy, V., Gao, R., and Li, Y. (2014). The DStar method for effective software fault localization. *IEEE Transactions on Reliability* 63 (1): 290–308.

198 Lucia, L., Lo, D., Jiang, L. et al. (2014). Extended comprehensive study of association measures for fault localization. *Journal of Software: Evaluation and Process* 26 (2): 172–219.

199 Janssen, T., Abreu, R., and van Germund, A.J.C. (2009). Zoltar: a spectrum-based fault localization tool. *Proceedings of the ESEC/FSE Workshop on Software Integration and Evaluation*, Amsterdam, Netherlands (August 2009), 23–30.

200 Abreu, R., Mayer, W., Stumptner, M., and van Gemund, A.J. (2009). Refining spectrum-based fault localization rankings. *Proceedings of the ACM Symposium on Applied Computing,* Honolulu, HI, USA (March 2009), 409–414.

201 Wong, W.E., Debroy, V., Golden, R. et al. (2012). Effective software fault localization using an RBF neural network. *IEEE Transactions on Reliability* 61 (1): 149–169.

202 Wong, W.E., Debroy, V., and Xu, D. (2012). Towards better fault localization: a crosstab-based statistical approach. *IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews* 42 (3): 378–396.

203 Wong, W.E., Debroy, V., Li, Y., and Gao, R. (2012). Software fault localization using DStar (D∗). *Proceedings of the Sixth International Conference on Software Security and Reliability*, Gaithersburg, MD, USA (June 2012), 21–30.

204 Kochhar, P.S., Xia, X., Lo, D., and Li, S. (2016). Practitioners' expectations on automated fault localization. *Proceedings of the 25th International Symposium on Software Testing and Analysis*, Saarbrücken, Germany (July 2016), 165–176.

205 Sun, S. and Podgurski, A. (2016). Properties of effective metrics for coverage-based statistical fault localization. *Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation*, Chicago, IL, USA (April 2016), 124–134.

206 Yoo, S., Xie, X., Kuo, F. et al. (2014). No Pot of Gold at the End of Program Spectrum Rainbow: Greatest Risk Evaluation Formula Does Not Exist. *Research Note RN/14/14*. University College London.

207 Ernst, M.D., Cockrell, J., Griswold, W.G., and Notkin, D. (2001). Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27 (2): 99–123.

208 Pytlik, B., Renieris, M., Krishnamurthi, S., and Reiss, S.P. (2003). Automated fault localization using potential invariants. arXiv preprint cs/0310040.

209 Sahoo, S.K., Criswell, J., Geigle, C., and Adve, V. (2013). Using likely invariants for automated software fault localization. *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Houston, TX, USA (March 2013), 139–152.

210 Alipour, M.A. and Groce, A. (2012). Extended program invariants: applications in testing and fault localization. *Proceedings of the Ninth International Workshop on Dynamic Analysis*, Minneapolis, MN, USA (July 2012), 7–11.

211 Shu, T., Ye, T., Ding, Z., and Xia, J. (2016). Fault localization based on statement frequency. *The Information of the Science* 360: 43–56.

212 Le, T.B., Lo, D., Le Goues, C., and Grunske, L. (2016). A learning-to-rank based fault localization approach using likely invariants. *Proceedings of the 25th International Symposium on Software Testing and Analysis*, Saarbrücken, Germany (July 2016), 177–188.

213 Liblit, B., Naik, M., Zheng, A.X. et al. (2005). Scalable statistical bug isolation. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementations*, Chicago, IL, USA (June 2005), 15–26.

214 Liu, C., Fei, L., Yan, X. et al. (2006). Statistical debugging: a hypothesis testing-based approach. *IEEE Transactions on Software Engineering* 32 (10): 831–848.

215 Dallmeier, V., Lindig, C., and Zeller, A. (2005). Lightweight defect localization for java. *Proceedings of the European Conference on Object-Oriented Programming*, Glasgow, UK (July 2005), 528–550.

216 Liu, C., Yan, X., Yu, H. et al. (2005). Mining behavior graphs for "Backtrace" of noncrashing bugs. *Proceedings of the SIAM International Conference on Data Mining*, Newport Beach, CA, USA (April 2005), 286–297.

217 Yilmaz, C., Paradkar, A., and Williams, C. (2008). Time will tell: fault localization using time spectra. *Proceedings of the International Conference on Software Engineering (ICSE '08)*, Leipzig, Germany (May 2008), 81–90.

218 Miraglia, A., Vogt, D., Bos, H. et al. (2016). Peeking into the past: efficient checkpoint-assisted time-traveling debugging. *Proceedings of the 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE '16)*, Ottawa, ON, Canada (23–27 October 2016), 455–466. IEEE. https://doi.org/10.1109/ISSRE.2016.9.

219 Treffer, A. and Uflacker, M. (2017). Back-in-time debugging in heterogeneous software stacks. *Proceedings of the 2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW '17)*, Toulouse, France (October 2017), 183–190. IEEE.

220 Xie, X., Chen, T.Y., and Xu, B. (2010). Isolating suspiciousness from spectrum-based fault localization techniques. *Proceedings of the International Conference on Quality Software*, Zhangjiajie, China (July 2010), 385–392.

221 Guo, L., Roychoudhury, A., and Wang, T. (2006). Accurately choosing execution runs for software fault localization. *Proceedings of the International Conference on Compiler Construction*, Vienna, Austria (March 2006), 80–95.

222 Abreu, R., González, A., and Gemund, A.J. (2010). Exploiting count spectra for bayesian fault localization. *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, Article No. 12, Timisoara, Romania (September 2010).

223 Lee, H.J., Naish, L., and Ramamohanarao, K. (2010). Effective software bug localization using spectral frequency weighting function. *Proceedings of the Annual IEEE International Computer Software and Applications Conference (COMPSAC '10)*, Seoul, Korea (July 2010), 218–227.

224 Xie, X., Wong, W.E., Chen, T.Y., and Xu, B. (2011). Spectrum-based fault localization: testing oracles are no longer mandatory. *Proceedings of the International Conference on Quality Software*, Madrid, Spain (July 2011), 1–10.

225 Xie, X., Wong, W.E., Chen, T.Y., and Xu, B. (2013). Metamorphic slice: an application in spectrum-based fault localization. *Information and Software Technology* 55 (5): 866–879.

226 Chen, T.Y., Tse, T.H., and Zhou, Z.Q. (2011). Semi-proving: an integrated method for program proving, testing, and debugging. *IEEE Transactions on Software Engineering* 37 (1): 109–125.

227 Tolksdorf, S., Lehmann, D., and Pradel, M. (2019). Interactive metamorphic testing of debuggers. *Proceedings of the 2019 ACM International Symposium on Software Testing and Analysis*, Beijing, China (July 2019), 273–283.

228 Zhao, L., Wang, L., Xiong, Z., and Gao, D. (2010). Execution-aware fault localization based on the control flow analysis. *Proceedings of the International Conference on Information Computing and Applications*, Tangshan, China (October 2010), 158–165.

229 Zhao, L., Wang, L., and Yin, X. (2011). Context-aware fault localization via control flow analysis. *Journal of Software* 6 (10): 1977–1984.

230 Guo, Y., Zhang, X., and Zheng, Z. (2016). Exploring the instability of spectra based fault localization performance. *Proceedings of the IEEE 40th Annual Computer Software and Applications Conference (COMPSAC '16)*, Atlanta, GA, USA (June 2016), 191–196.

231 Keller, F., Grunske, L., Heiden, S. et al. (2017). A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. *Proceedings of the IEEE International Conference on Software Quality, Reliability, and Security (QRS '17)*, Prague, Czech Republic (July 2017), 114–125.

232 Li, Z., Yan, L., Liu, Y. et al. (2018). MURE: making use of mutations to refine spectrum-based fault localization. *Proceedings of the IEEE International Conference on Software Quality, Reliability, and Security Companion (QRS-C '18)*, Lisbon, Portugal (July 2018), 56–63.

233 Pearson, S. (2016). Evaluation of fault localization techniques. *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Seattle, WA, USA (November 2016), 1115–1117.

234 Pearson, S., Campos, J., Just, R. et al. (2017). Evaluating and improving fault localization. *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE '17)*, Buenos Aires, Argentina (May 2017), 609–620.

235 de Oliveira, A.A.L., Camilo-Junior, C.G., de Andrade Freitas, E.N., and Vincenzi, A.M.R. (2018). FTMES: a failed-test-oriented mutant execution strategy for mutation-based fault localization. *Proceedings of the 2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE '18)*, Memphis, TN, USA (15–18 October 2018), 155–165. IEEE. https://doi.org/10.1109/ISSRE.2018.00026.

236 Liu, Y., Li, Z., Wang, L. et al. (2017). Statement-oriented mutant reduction strategy for mutation based fault localization. *Proceedings of the IEEE International Conference on Software Quality, Reliability, and Security (QRS '17)*, Prague, Czech Republic (July 2017), 126–137.

237 Jeon, J. and Hong, S. (2020). Threats to validity in experimenting mutation-based fault localization. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20): New Ideas and Emerging Results*, Seoul, South Korea (27 June to 19 July 2020), 1–4. ACM. https://dl.acm.org/doi/10.1145/3377816.3381746.

238 Li, A., Lei, Y., and Mao, X. (2016). Towards more accurate fault localization: an approach based on feature selection using branching execution probability. *Proceedings of the IEEE International Conference on Software Quality, Reliability, and Security (QRS '16)*, Vienna, Austria (August 2016), 431–438.

239 Perez, A., Abreu, R., and Riboira, A. (2014). A dynamic code coverage approach to maximize fault localization efficiency. *Journal of Systems and Software* 90: 18–28.

240 Chilimbi, T.M., Liblit, B., Mehra, K. et al. (2009). HOLMES: effective statistical debugging via efficient path profiling. *Proceedings of the IEEE International Conference on Software Engineering (ICSE '09)*, Vancouver, ON, Canada (May 2009), 34–44.

241 Hu, P., Zhang, Z., Chan, W.K., and Tse, T.H. (2008). Fault localization with non-parametric program behavior model. *Proceedings of the International Conference on Quality Software*, Oxford, UK (12–13 August 2008), 385–395.

242 Zhang, Z., Chan, W.K., Tse, T.H. et al. (2011). Non-parametric statistical fault localization. *Journal of Systems and Software* 84 (6): 885–905.

243 Yang, Y., Deng, F., Yan, Y., and Gao, F. (2019). A fault localization method based on conditional probability. *Proceedings of the IEEE International Conference on Software Quality, Reliability, and Security Companion (QRS-C '19)*, Sofia, Bulgaria (July 2019), 213–218

244 Henderson, T.A.D. and Podgurski, A. (2018). Behavioral fault localization by sampling suspicious dynamic control flow subgraphs. *Proceedings of the 2018 IEEE*

International Conference on Software Testing, Verification and Validation, Vasteras, Sweden (April 2018), 93–104.

245 Zhang, Z., Jiang, B., Chan, W.K., and Tse, T.H. (2008). Debugging through evaluation sequences: a controlled experimental study. *Proceedings of the International Computer Software and Applications Conference (COMPSAC '08)*, Turku, Finland (28 July to 1 August 2008), 128–135.

246 Zhang, Z., Jiang, B., Chan, W.K. et al. (2010). Fault localization through evaluation sequences. *Journal of Systems and Software* 83 (2): 174–187.

247 You, Z., Qin, Z., and Zheng, Z. (2012). Statistical fault localization using execution sequence. *Proceedings of the International Conference on Machine Learning and Cybernetics*, Xi'an, China (July 2012), 899–905.

248 Baah, G.K., Podgurski, A., and Harrold, M.J. (2010). Causal inference for statistical fault localization. *Proceedings of the International Symposium on Software Testing and Analysis*, Trento, Italy (July 2010), 73–83.

249 Baah, G.K., Podgurski, A., and Harrold, M.J. (2011). Mitigating the confounding effects of program dependences for effective fault localization. *Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Szeged, Hungary (September 2011), 146–156.

250 Modi, V., Roy, S., and Aggarwal, S.K. (2013). Exploring program phases for statistical bug localization. *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, Seattle, WA, USA (June 2013), 33–40.

251 Wang, X., Jiang, S., Ju, X. et al. (2015). Mitigating the dependence confounding effect for effective predicate-based statistical fault localization. *Proceedings of the IEEE 39th Annual Computer Software and Applications Conference (COMPSAC '15)*, Taichung, Taiwan (July 2015), 105–114.

252 Feyzi, F. and Parsa, S. (2018). FPA-FL: incorporating static fault-proneness analysis into statistical fault localization. *Journal of Systems and Software* 136: 39–58.

253 Abramson, D., Foster, I., Michalakes, J., and Sosic, R. (1995). Relative debugging and its application to the development of large numerical models. *Proceedings of the 8th International Conference for High Performance Computing, Networking, Storage, and Analysis*, no. 51, San Diego, CA, USA (December 1995).

254 Zeller, A. and Hildebrandt, R. (2002). Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28 (2): 183–200.

255 Zimmermann, T. and Zeller, A. (2001). Visualizing memory graphs. In: *Software Visualization* (ed. Diehl, S.), 191–204. Berlin, Heidelberg: Springer.

256 Wikipedia. Delta debugging. https://en.wikipedia.org/wiki/Delta_debugging (accessed January 2022).

257 Nie, C. and Leung, H. (2011). The minimal failure-causing schema of combinatorial testing. *ACM Transactions on Software Engineering and Methodology* 20 (4): 1–38.

258 Niu, X., Nie, C., Lei, Y., and Chan, A.T., Identifying failure-inducing combinations using tuple relationship. *Proceedings of the International Conference on Software Testing, Verification, and Validation Workshops*, Luxembourg (March 2013), 271–280.

259 Ghandehari, L.S., Lei, Y., Kacker, R. et al. (2018). A combinatorial testing-based approach to fault localization. *IEEE Transactions on Software Engineering* 46 (6): 616–645.

260 Gupta, N., He, H., Zhang, X., and Gupta, R. (2005). Locating faulty code using failure-inducing chops. *Proceedings of the International Conference on Automated Software Engineering*, Long Beach, CA, USA (November 2005), 263–272.

261 Sumner, W.N and Zhang, X. (2010). Memory indexing: canonicalizing addresses across executions. *Proceedings of the International Symposium on Foundations of Software Engineering*, Santa Fe, NM, USA (November 2010), 217–226.

262 Sumner, W.N. and Zhang, X. (2009). Algorithms for automatically computing the causal paths of failure. *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, York, UK (March 2009), 335–369.

263 Xin, B., Sumner, W.N., and Zhang, X. (2008). Efficient program execution indexing. *ACM SIGPLAN Notices* 43 (6): 238–248.

264 Sumner, W.N. and Zhang, X. (2013). Comparative causality: explaining the differences between executions. *Proceedings of the International Conference on Software Engineering (ICSE '13)*, San Francisco, CA, USA (May 2013), 272–281.

265 Hashimoto, M., Mori, A., and Izumida, T. (2018). Automated patch extraction via syntax-and semantics-aware delta debugging on source code changes. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Lake Buena Vista, FL, USA (November 2018), 598–609.

266 Zhang, X., Gupta, N., and Gupta, R. (2006). Locating faults through automated predicate switching. *Proceedings of the International Conference on Software Engineering (ICSE '06)*, Shanghai, China (May 2006), 272–281.

267 Wang, T. and Roychoudhury, A. (2005). Automated path generation for software fault localization. *Proceedings of the International Conference on Automated Software Engineering*, Long Beach, CA, USA (November 2005), 347–351.

268 Li, F., Huo, W., Chen, C. et al. (2013). Effective fault localization based on minimum debugging frontier set. *Proceedings of the International Symposium on Code Generation and Optimization*, Shenzhen, China (February 2013), 1–10.

269 Li, F., Li, Z., Huo, W., and Feng, X. (2017). Locating software faults based on minimum debugging frontier set. *IEEE Transactions on Software Engineering* 43 (8): 760–776.

270 Jeffrey, D., Gupta, N. and Gupta, R. (2008). Fault localization using value replacement. *Proceedings of the International Symposium on Software Testing and Analysis*, Seattle, WA, USA (July 2008), 167–178.

271 Zhang, Z., Chan, W.K., Tse, T.H. et al. (2009). Capturing propagation of infected program states. *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundation of Software Engineering*, Amsterdam, The Netherlands (24–28 August 2009), 43–52.

272 Yang, B., He, Y., Liu, H. et al. (2020). A lightweight fault localization approach based on XGBoost. *Proceedings of the 2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS '20)*, Macau, China (11–14 December 2020), 168–179. IEEE. https://doi.org/10.1109/QRS51102.2020.00033.

273 Zhong, H. and Mei, H. (2020). Learning a graph-based classifier for fault localization. *Science China Information Sciences* 63 (6): 1–22.

274 Nishiura, K., Choi, E., and Mizuno, O. (2017). Improving faulty interaction localization using logistic regression. *Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS '17)*, Prague, Czech Republic (25–29 July 2017), 138–149. IEEE. https://doi.org/10.1109/QRS.2017.24.

275 Xiao, Y., Keung, J., Bennin, K.E., and Mi, Q. (2019). Improving bug localization with word embedding and enhanced convolutional neural networks. *Information and Software Technology* 105: 17–29.

276 Xiao, Y., Keung, J., Bennin, K.E., and Mi, Q. (2018). Machine translation-based bug localization technique for bridging lexical gap. *Information and Software Technology* 99: 58–61.

277 Wong, W.E. and Qi, Y. (2009). BP neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering* 19 (4): 573–597.

278 Fausett, L. (1994). *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*. Prentice Hall.

279 Hecht-Nielsen, R. (1989). Theory of the backpropagation neural network. *Proceedings of the International Joint Conference on Neural Networks*, Washington DC, USA (June 1989), 593–605. IEEE.

280 Lee, C.C., Chung, P.C., Tsai, J.R., and Chang, C.I. (1999). Robust radial basis function neural networks. *IEEE Transactions on Systems, Man, and Cybernetics: Part B Cybernetics* 29 (6): 674–685.

281 Wasserman, P.D. (1993). *Advanced Methods in Neural Computing*. Van Nostrand Reinhold.

282 Briand, L.C., Labiche, Y., and Liu, X. (2007). Using machine learning to support debugging with tarantula. *Proceedings of the IEEE International Symposium on Software Reliability*, Trolhattan, Sweden (November 2007), 137–146.

283 Jonsson, L., Broman, D., Magnusson, M. et al. (2016). Automatic localization of bugs to faulty components in large scale software systems using bayesian

classification. *Proceedings of the IEEE International Conference on Software Quality, Reliability, and Security (QRS '16)*, Vienna, Austria (August 2016), 423–430.

284 Mariani, L., Monni, C., Pezzé, M. et al. (2018). Localizing faults in cloud systems. *Proceedings of the 2018 IEEE International Conference on Software Testing, Verification and Validation*, Vasteras, Sweden (April 2018), 262–273.

285 Kim, Y., Mun, S., Yoo, S., and Kim, M. (2019). Precise learn-to-rank fault localization using dynamic and static features of target programs. *ACM Transactions on Software Engineering and Methodology* 28 (4) Article 23.

286 Zhang, M., Li, X., Zhang, L., and Khurshid, S. (2017). Boosting spectrum-based fault localization using pagerank. *Proceedings of the 2017 ACM International Symposium on Software Testing and Analysis*, Santa Barbara, CA, USA (July 2017), 261–272.

287 Sohn, J. (2020). Bridging fault localisation and defect prediction. *Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '20)*, Seoul, South Korea (27 June to 19 July 2020), 214–217. ACM. https://dl.acm.org/doi/10.1145/3377812.3381403.

288 Sohn, J. and Yoo, S. (2017). FLUCCS: using code and change metrics to improve fault localization. *Proceedings of the 2017 ACM International Symposium on Software Testing and Analysis*, Santa Barbara, CA, USA (July 2017), 273–283.

289 Li, X., Li, W., Zhang, Y., and Zhang, L. (2019). DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. *Proceedings of the 2019 ACM International Symposium on Software Testing and Analysis*, Beijing, China (July 2019), 169–180.

290 Near, J.P. and Jackson, D. (2016). Finding security bugs in web applications using a catalog of access control patterns. *Proceedings of the 2016 IEEE/ACM 38th IEEE International Conference on Software Engineering (ICSE '16)*, Austin, TX, USA (14–22 May 2016), 947–958. ACM. https://dl.acm.org/doi/10.1145/2884781.2884836.

291 Santos, D., Rodrigues, C.A., and Matias, R. Jr. (2018). Failure patterns in operating systems: an exploratory and observational study. *Journal of Systems and Software* 137: 512–530.

292 Zhou, X., Peng, X., Xie, T. et al. (2019). Latent error prediction and fault localization for microservice applications by learning from system trace logs. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Tallinn, Estonia (26–30 August 2019), 683–694. ACM. https://dl.acm.org/doi/10.1145/3338906.3338961.

293 Nessa, S., Abedin, M., Wong, W.E. et al. (2009). Fault localization using N-gram analysis. *Proceedings of the Third International Conference on Wireless Algorithms, Systems, and Applications*, Dallas, TX (26–28 October 2008), 548–559. ACM. https://doi.org/10.1007/978-3-540-88582-5_51.

294 Cellier, P., Ducasse, M., Ferre, S., and Ridoux, O. (2008). Formal concept analysis enhances fault localization in software. *Proceedings of the International Conference on Formal Concept Analysis*, Montreal, QC, Canada (February 2008), 273–288.

295 Cellier, P., Ducasse, M., Ferre, S., and Ridoux, O. (2011). Multiple fault localization with data mining. *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*, Miami, FL, USA (July 2011), 238–243.

296 Zhang, S. and Zhang, C. (2014). Software bug localization with Markov logic. *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion '14)*, Hyderabad, India (31 May to 7 June 2014), 424–427.

297 Denmat, T., Ducassé, M., and Ridoux, O. (2005). Data mining and cross-checking of execution traces. *Proceedings of the International Conference on Automated Software Engineering*, Long Beach, CA, USA (November 2005), 396–399.

298 Bian, P., Liang, B., Zhang, Y. et al. (2019). Detecting bugs by discovering expectations and their violations. *IEEE Transactions on Software Engineering* 45 (10): 984–1001.

299 Hanam, Q., de Brito, F.S.M., and Mesbah, A. (2016). Discovering bug patterns in JavaScript. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Seattle, WA, USA (13–18 November 2016), 144–156. ACM. https://dl.acm.org/doi/10.1145/2950290.2950308.

300 Reiter, R. (1987). A theory of diagnosis from first principles. *Artificial Intelligence* 32 (1): 57–95.

301 Mayer, W. and Stumptner, M. (2007). Model-based debugging: state of the art and future challenges. *Electronic Notes in Theoretical Computer Science* 174 (4): 61–82.

302 Mayer, W. and Stumptner, M. (2008). Evaluating models for model-based debugging. *Proceedings of the ACM International Conference on Automated Software Engineering*, L'Aquila, Italy (September 2008), 128–137.

303 Abreu, R. and van Gemund, A.J.C. (2009). A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. *Proceedings of the Eight Symposium on Abstraction, Reformulation, and Approximation*, Lake Arrowhead, CA, USA (August 2009).

304 Baah, G.K., Podgurski, A., and Harrold, M.J. (2010). The probabilistic program dependence graph and its application to fault diagnosis. *IEEE Transactions on Software Engineering* 36 (4): 528–545.

305 DeMillo, R.A., Pan, H., and Spafford, E.H. (1997). Failure and fault analysis for software debugging. *Proceedings of the Twenty-First Annual International Computer Software and Applications Conference (COMPSAC '97)*, Washington, DC, USA (11–15 August 1997), 515–521. IEEE. https://doi.org/10.1109/CMPSAC.1997.625061.

306 Friedrich, G., Stumptner, M., and Wotawa, F. (1999). Model-based diagnosis of hardware designs. *Artificial Intelligence* 1–2: 3–39.

307 de Kleer, J. and Williams, B.C. (1987). Diagnosing multiple faults. *Artificial Intelligence* 32 (1): 97–130.

308 Mateis, C., Stumptner, M., and Wotawa, F. (2000). Modeling java programs for diagnosis. *Proceedings of the 14th European Conference on Artificial Intelligence*, Berlin, Germany (20–25 August 2000), 171–175. IOS Press.

309 Mayer, W., Abreu, R., Stumptner, M., and van Gemund, A.J.C. (2008). Prioritizing model-based debugging diagnostic reports. *Proceedings of the International Workshop on Principles of Diagnosis*, Blue Mountains, Australia (September 2008), 127–134.

310 Mayer, W. and Stumptner, M. (2004). Approximate modeling for debugging of program loops.

311 Mayer, W. and Stumptner, M. (2003). Model-based debugging using multiple abstract models. arXiv preprint cs/0309030.

312 Shchekotykhin, K.M., Schmitz, T., and Jannach, D. (2016). Efficient sequential model-based fault-localization with partial diagnose. *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, New York City, NY, USA (July 2016), 1251–1257.

313 Wotawa, F., Stumptner, M., and Mayer, W. (2002). Model-based debugging or how to diagnose programs automatically. *Proceedings of the International Conference on Industrial and Engineering, Applications of Artificial Intelligence and Expert Systems*, Cairns, Australia (June 2002), 746–757.

314 Wotawa, F., Weber, J., Nica, M., and Ceballos, R. (2009). On the complexity of program debugging using constraints for modeling the program's syntax and semantics. *Proceedings of the Conference of the Spanish Association for Artificial Intelligence*, Seville, Spain (November 2009), 22–31.

315 Wu, J., Jia, X., Liu, C. et al. (2004). A statistical model to locate faults at input levels. *Proceedings of the International Conference on Automated Software Engineering*, Linz, Austria (September 2004), 274–277.

316 Mateis, C., Stumptner, M., Wieland, D., and Wotawa, F. (2002). JADE – AI support for debugging java programs. *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence*, Vancouver, BC, Canada (November 2002), 62.

317 Mayer, W., Stumptner, M., Wieland, D., and Wotawa, F. (2002). Towards an integrated debugging environment. *Proceedings of the European Conference on Artificial Intelligence*, Lyon, France (July 2002), 422–426.

318 Abreu, R., Zoeteweij, P., and van Gemund, A.J. (2009). Spectrum-based multiple fault localization. *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, Auckland, New Zealand (November 2009), 88–99.

319 Mayer, W. and Stumptner, M. (2002). Modeling programs with unstructured control flow for debugging. *Australian Joint Conference on Artificial Intelligence*, Canberra, Australia (December 2002), 107–118.

320 Mayer, W. and Stumptner, M. (2003). Extending diagnosis to debug programs with exceptions. *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, Montreal, QC, Canada (6–10 October 2003), 240–244. IEEE. https://doi.org/10.1109/ASE.2003.1240312.

321 Xu, Z., Ma, S., Zhang, X. et al. (2018). Debugging with the intelligence via probabilistic inference. *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE '18)*, Gothenburg, Sweden (May 2018), 1171–1181.

322 Yu, X., Liu, J., Yang, Z.J. et al. (2016). Bayesian network based program dependence graph for fault localization. *Proceedings of the 2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW '16)*, Ottawa, ON, Canada (23–27 October 2016), 181–188. IEEE. https://doi.org/10.1109/ISSREW.2016.35.

323 Yu, X., Liu, J., Yang, Z., and Liu, X. (2017). The Bayesian network based program dependence graph and its application to fault localization. *Journal of Systems and Software* 134: 44–53.

324 Bourdoncle, F. (1993). Abstract debugging of higher-order imperative languages. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation,* Albuquerque, NM, USA (June 1993), 46–55.

325 Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of the IEEE International Symposium on Principles of Programming Languages*, Los Angeles, CA, USA (January 1977), 238–252.

326 Mayer, W. and Stumptner, M. (2007). Abstract interpretation of programs for model-based debugging. *Proceedings of the International Joint Conference on Artificial Intelligence*, Hyderabad, India (January 2007), 471–476.

327 Kob, D. and Wotawa, F. (2004). Introducing alias information into model-based debugging. *Proceedings of the European Conference on Artificial Intelligence*, Valencia Spain (August 2004), 833–837

328 Mayer, W., Stumptner, M., Wieland, D., and Wotawa, F. (2002). Can AI help to improve debugging substantially? Debugging experiences with value-based models. *Proceedings of the European Conference on Artificial Intelligence*, Lyon, France (July 2002), 417–421

329 Ball, T., Naik, M., and Rajamani, S.K. (2003). From symptom to cause: localizing errors in counterexample traces. *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New Orleans, LA, USA (January 2003), 97–105.

330 Chaki, S., Groce, A., and Strichman, O. (2004). Explaining abstract counterexamples. *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, Newport Beach, CA, USA (31 October to 6 November 2004), 73–82. ACM.

331 Griesmayer, A., Staber, S., and Bloem, R. (2007). Automated fault localization for C programs. *Electronic Notes in Theoretical Computer Science* 174 (4): 95–111.

332 Griesmayer, A., Staber, S., and Bloem, R. (2010). Fault localization using a model checker. *Software Testing, Verification and Reliability* 20 (2): 149–173.

333 Groce, A. and Visser, W. (2003). What went wrong: explaining counterexamples. *Proceedings of the International Conference on Model Checking Software*, Portland, OR, USA (May 2003), 121–136.

334 Groce, A., Kroening, D., and Lerda, F. (2004). Understanding counterexample with explain. *International Conference on Computer Aided Verification*, Boston, MA, USA (July 2004), 453–456.

335 Könighofer, R. and Bloem, R. (2011). Automated error localization and correction for imperative programs. *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, Austin, TX, USA (October 2011), 91–100.

336 da Alves, E.H.S., Cordeiro, L.C., and Eddiefilho, B.d.L. (2017). A method to localize faults in concurrent C programs. *Journal of Systems and Software* 132: 336–352.

337 Debroy, V. and Wong, W.E. (2014). Combining mutation and fault localization for automated program debugging. *Journal of Systems and Software* 90: 45–60.

338 He, H. and Gupta, N. (2004). Automated debugging using path-based weakest preconditions. *Proceedings of the Fundamental Approaches to Software Engineering*, Barcelona, Spain (March 2004), 267–280.

339 Kaleeswaran, S., Tulsian, V., Kanade, A., and Orso, A. (2014). MintHint: automated synthesis of repair hints. *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*, Hyderabad, India (31 May to 7 June 2014), 266–276. ACM. https://dl.acm.org/doi/10.1145/2568225.2568258.

340 Nguyen, H., Qi, D., Roychoudhury, A., and Chandra, S. (2013). SemFix: program repair via semantic analysis. *Proceedings of the International Conference on Software Engineering (ICSE '13)*, San Francisco, CA, USA (May 2013), 772–781.

341 Stumptner, M. and Wotawa, F. (1999). Debugging functional program. *Proceedings of the International Joint Conference on Artificial Intelligence*, Stockholm, Sweden (31 July to 6 August 1999), 1074–1079.

342 Peischl, B. and Wotawa, F. (2006). Automated source-level error localization in hardware designs. *IEEE Design and Test of Computers* 23 (1): 8–19.

343 Wotawa, F. (2002). Debugging hardware designs using a value-based model. *Applied Intelligence* 16 (1): 71–92.

344 Jannach, D. and Engler, U. (2010). Toward model-based debugging of spreadsheet programs. *Proceedings of the Joint Conference on Knowledge-Based Software Engineering*, Kaunas, Lithuania (August 2010), 252–264.

345 Nica, M., Nica, S., and Wotawa, F. (2013). On the use of mutations and testing for debugging. *Software, Practice and Experience* 43 (9): 1121–1142.

346 Wotawa, F., Nica, M., and Moraru, I. (2012). Automated debugging based on a constraint model of the program and a test case. *The Journal of Logic and Algebraic Programming* 81 (4): 390–407.

347 Soria, A., Pace, J.A.D., and Campo, M.R. (2015). Architecture-driven assistance for fault-localization tasks. *Expert Systems* 32 (1): 1–22.

348 Chittimalli, P.K. and Shah, V. (2015). Fault localization during system testing. *Proceedings of the IEEE 23rd International Conference on Program Comprehension*, Florence, Italy (May 2015), 285–286.

349 Lehmann, D. and Pradel, M. (2018). Feedback-directed differential testing of interactive debuggers. *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Lake Buena Vista, FL, USA (November 2018), 610–620.

350 Wang, Y., Li, J., Yan, N. et al. (2018). Bug patterns localization based on topic model for bugs in program loop. *Proceedings of the IEEE International Conference on Software Quality, Reliability, and Security Companion (QRS-C '18)*, Lisbon, Portugal (July 2018), 366–370.

351 Troya, J., Segura, S., Parejo, J.A., and Ruiz-Cortés, A. (2018). Spectrum-based fault localization in model transformations. *ACM Transactions on Software Engineering and Methodology* 27 (3): 1–50, Article 13.

352 Bohnet, J., Voigt, S., and Döllner, J. (2009). Projecting code changes onto execution traces to support localization of recently introduced bugs. *Proceedings of the ACM Symposium on Applied Computing,* Honolulu, HI, USA (March 2009), 438–442.

353 Chen, K. and Chen, J. (2007). Aspect-based instrumentation for locating memory leaks in java programs. *Proceedings of the Annual International Computer Software and Applications Conference (COMPSAC '07)*, Beijing, China (July 2007), 23–28.

354 Chen, I., Yang, C., Lu, T., and Jaygarl, H. (2008). Implicit social network model for predicting and tracking the location of faults. *Proceedings of the Annual International Computer Software and Applications Conference (COMPSAC '08)*, Turku, Finland (August 2008), 136–143.

355 de Souza, H.A. and Chaim, M.L. (2013). Adding context to fault localization with integration coverage. *Proceedings of the International Conference on Automated Software Engineering*, Silicon Valley, CA, USA (November 2013), 628–633.

356 Gong, L., Lo, D., Jiang, L., and Zhang, H. (2012). Interactive fault localization leveraging simple user feedback. *Proceedings of the International Conference on Software Maintenance*, Trento, Italy (September 2012), 67–76.

357 Lin, Y., Sun, J., Xue, Y. et al. (2017). Feedback-based debugging. *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE '17)*, Buenos Aires, Argentina (May 2017), 393–403.

358 Li, X., Zhu, S., d'Amorim, M., and Orso, A. (2018). Enlightened debugging. *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE '18)*, Gothenburg, Sweden (May 2018), 82–92.

359 Ko, A.J. and Myers, B.A. (2008). Debugging reinvented: asking and answering why and why not questions about program behavior. *Proceedings of the International Conference on Software Engineering (ICSE '08)*, Leipzig, Germany (May 2008), 301–310.

360 Cheng, H., Lo, D., Zhou, Y. et al. (2009). Identifying bug signatures using discriminative graph mining. *Proceedings of the International Symposium on Software Testing and Analysis*, Chicago, IL, USA (July 2009), 141–152.

361 Maruyama, K. and Terada, M. (2003). Debugging with reverse watchpoint. *Proceedings of the International Conference on Quality Software,* Dallas, TX, USA (November 2003), 116–116.

362 Liu, B., Nejati, L.S., Briand, L.C., and Bruckmann, T. (2016). Simulink fault localization: an iterative statistical debugging approach. *Software Testing, Verification and Reliability* 26 (6): 431–459.

363 Liu, B., Nejati, S., and Briand, L.C. (2019). Effective fault localization of automotive Simulink models: achieving the trade-off between test oracle effort and fault localization accuracy. *Empirical Software Engineering* 24 (1): 444–490.

364 Wang, Q., Parnin, C., and Orso, A. (2015). Evaluating the usefulness of IR-based fault localization techniques. *Proceedings of the 24th International Symposium on Software Testing and Analysis*, Baltimore, MD, USA (July 2015), 1–11.

365 Dallmeier, V. and Zimmermann, T. (2007). Extraction of bug localization benchmarks from history. *Proceedings of the International Conference on Automated Software Engineering*, Atlanta, GA, USA (November 2007), 433–436.

366 Lee, J., Kim, D., and Bissyandé, T.F. (2018). Bench4BL: reproducibility study on the performance of IR-based bug localization. *Proceedings of the 2018 ACM International Symposium on Software Testing and Analysis*, Amsterdam, Netherlands (July 2018), 61–72.

367 Lukins, S.K., Kraft, N.A., and Etzkorn, L.H. (2008). Source code retrieval for bug localization using latent dirichlet allocation. *Proceedings of the Working Conference on Reverse Engineering*, Antwerp, Belgium (October 2008), 155–164.

368 Rao, S. and Kak, A. (2011). Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. *Proceedings of the Working Conference on Mining Software Repositories*, Honolulu, HI, USA (May 2011), 43–52.

369 Saha, R.K., Lease, M., Kunshid, S., and Perry, D.E. (2013). Improving bug localization using structured information retrieval. *Proceedings of the International Conference on Automated Software Engineering*, Silicon Valley, CA, USA (November 2013), 345–355.

370 Zhou, J., Zhang, H., and Lo, D. (2012). Where should the bugs be fixed? – More accurate information retrieval-based bug localization based on bug reports. *Proceedings of the International Conference on Software Engineering (ICSE '12),* Zurich, Switzerland (June 2012), 14–24.

371 Chaparro, O., Florez, J.M., and Marcus, A. (2019). Using bug descriptions to reformulate queries during text-retrieval-based bug localization. *Empirical Software Engineering* 24 (5): 2947–3007.

372 Tantithamthavorn, C., Abebe, S.L., Hassan, A.E. et al. (2018). The impact of IR-based classifier configuration on the performance and the effort of method-level bug localization. *Information and Software Technology* 102: 160–174.

373 Khatiwada, S., Tushev, M., and Mahmoud, A. (2018). Just enough semantics: an information theoretic approach for IR-based software bug localization. *Information and Software Technology* 93: 45–57.

374 Zhang, L. and Zhang, Z. (2018). SeTCHi: selecting test cases to improve history-guided fault localization. *Proceedings of the 2018 IEEE 29th International Symposium on Software Reliability Engineering Workshops (ISSREW '18)*, Memphis, TN, USA (15–18 October 2018), 200–207. IEEE. https://doi.org/10.1109/ISSREW.2018.00007.

375 Rahman, M.M. and Roy, C.K. (2018). Improving IR-based bug localization with context-aware query reformulation. *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Lake Buena Vista, FL, USA (November 2018), 621–632.

376 Amar, A. and Rigby, P.C. (2019). Mining historical test logs to predict bugs and localize faults in the test logs. *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE '19)*, Montréal, QC, Canada (May 2019), 140–151.

377 Le, T.B., Thung, F., and Lo, D. (2017). Will this localization tool be effective for this bug? Mitigating the impact of unreliability of information retrieval based bug localization tools. *Empirical Software Engineering* 22: 2237–2279.

378 Hoang, T., Oentary, R.J., Le, T.B., and Lo, D. (2019). Network-clustered multimodal bug localization. *IEEE Transactions on Software Engineering* 45 (10): 1002–1023.

379 Silva, J. (2011). A survey on algorithmic debugging strategies. *Advances in Engineering Software* 42 (11): 976–991.

380 Zeller, A. (2006). *Why Programs Fail – A Guide to Systematic Debugging*. Elsevier.

381 Christ, J., Ermis, E., Schaf, M. and Wies, T. (2013). Flow-sensitive fault localization. *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation*, Rome, Italy (January 2013), 189–208.

382 Ermis, E., Schaf, M., and Wies, T. (2012). Error invariants. *Proceedings of the International Symposium on Formal Methods*, Paris, France (August 2012), 187–201.

383 Jose, M. and Majumdar, R. (2011). Bug-assist: assisting fault localization in ANSI-C programs. *Proceedings of the International Conference on Computer Aided Verification*, Snowbird, UT, USA (July 2011), 504–509.

384 Jose, M. and Majumdar, R. (2011). Cause clue clauses: error localization using maximum satisfiability. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Jose, CA, USA (June 2011), 437–446.

385 Lamraoui, S. and Nakajima, S. (2016). A formula-based approach for automatic fault localization of multi-fault programs. *Journal of Information Processing* 24 (1): 88–98.

386 Roychoudhury, A. and Chandra, S. (2016). Formula-based software debugging. *Communications of the ACM* 59 (7): 68–77.

387 Zhao, F., Tang, Y., Yang, Y. et al. (2015). Is learning-to-rank cost-effective in recommending relevant files for bug localization? *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security (QRS '15)*, Vancouver, BC, Canada (August 2015), 293–303.

388 Wu, R., Wen, M., Cheung, S., and Zhang, H. (2018). ChangeLocator: locate crash-inducing changes based on crash reports. *Empirical Software Engineering* 23: 2866–2900.

389 Chen, H., Lv, H., Huang, S. et al. (2015). Diagnosing SDN network problems by using spectrum-based fault localization techniques. *Proceedings of the IEEE International Conference on Software Quality, Reliability, and Security Companion (QRS-C '15)*, Vancouver, BC, Canada (August 2015), 121–127.

390 Christi, A., Groce, A., and Gopinath, R. (2019). Evaluating fault localization for resource adaptation via test-based software modification. *Proceedings of the IEEE International Conference on Software Quality, Reliability, and Security (QRS '19)*, Sofia, Bulgaria (July 2019), 26–33.

391 Andrews, J.H., Briand, L.C., and Labiche, Y. (2005). Is mutation an appropriate tool for testing experiments? *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, St. Louis, MO, USA (May 2005), 402–411.

392 Do, H. and Rothermel, G. (2006). On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering* 32 (9): 733–752.

393 Namin, A.S., Andrews, J.H., and Labiche, Y. (2006). Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering* 32 (8): 608–624.

394 Liu, C. and Han, J. (2006). Failure proximity: a fault localization-based approach. *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Portland, OR, USA (November 2006), 46–56.

395 Hofer, B. and Wotawa, F. (2012). Spectrum enhanced dynamic slicing for better fault localization. *Proceedings of the European Conference on Artificial Intelligence*, Montpellier, France (August 2012), 420–425.

396 Parnin, C. and Orso, A. (2011). Are automated debugging techniques actually helping programmers? *Proceedings of the International Symposium on Software Testing and Analysis*, Toronto, ON, Canada (July 2011), 199–209.

397 Xie, X., Liu, Z., Song, S. et al. (2016). Revisit of automatic debugging via human focus-tracking analysis. *Proceedings of the International Conference on Software Engineering (ICSE '16)*, Austin, TX, USA (May 2016), 808–819.

398 Xia, X., Bao, L., Lo, D., and Li, S. (2017). "Automated debugging considered harmful" considered harmful: a user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. *Proceedings of the International Conference on Software Maintenance and Evolution*, Raleigh, NC, USA (January 2017), 267–278.

399 Le, T.B., Lo, D., and Thung, F. (2015). Should I follow this fault localization tool's output? – Automated prediction of fault localization effectiveness. *Empirical Software Engineering* 20 (5): 1237–1274.

400 Monperrus, M. (2014). A critical review of "automated patch generation learned from human-written patches" essay on the problem statement and the evaluation of automatic software repair. *Proceedings of the International Conference on Software Engineering (ICSE '14)*, Hyderabad, India (June 2014), 234–242.

401 Debroy, V. and Wong, W.E. (2009). Insights on fault interference for programs with multiple bugs. *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE '09)*, Karnataka, India (November 2009), 165–174.

402 Steimann, F., Frenkel, M., and Abreu, R. (2013). Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. *Proceedings of the International Symposium on Software Testing and Analysis*, Lugano, Switzerland (July 2013), 314–324.

403 Artzi, S., Dolby, J., Tip, F., and Pistoia, M. (2010). Practical fault localization for dynamic web applications. *Proceedings of the IEEE International Conference on Software Engineering (ICSE '10)*, Cape Town, South Africa (May 2010), 265–274.

404 Flanagan, C, Freund, S. N., and Yi, J. (2008). Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Tucson, AZ, USA (June 2008), 293–303.

405 Gulzar, M.A., Interlandi, M., Condie, T., and Kim, M. (2016). BigDebug: debugging primitives for interactive big data processing in spark. *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE '16)*, Austin, TX, USA (May 2016), 784–795.

406 Gulzar, M.A., Interlandi, M., Yoo, S. et al. (2016). BigDebug: interactive debugger for big data analytics in apache spark. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Seattle, WA, USA (13–18 November 2016), 1033–1037. ACM. https://dl.acm.org/doi/10.1145/2950290.2983930.

407 Gulzar, M.A., Wang, S., and Kim, M. (2018). BigSift: automated debugging of big data analytics in data-intensive scalable computing. *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Lake Buena Vista, FL, USA (November 2018), 863–866.

408 Jeffrey, D., Feng, M., Gupta, N., and Gupta, R. (2009). BugFix: a learning-based tool to assist developers in fixing bugs. *Proceedings of the International Conference on Program Comprehension*, Vancouver, BC, Canada (May 2009), 70–79.

409 Yang, Y., Zhou, Y., Sun, H. et al. (2019). Hunting for bugs in code coverage tools via randomized differential testing. *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE '19)*, Montréal, QC, Canada (May 2019), 488–498.

410 Naik, M., Aiken, A., and Whaley, J. (2006). Effective static race detection for java. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Ottawa, ON, Canada (June 2006), 308–319.

411 Csallner, C. and Smaragdakis, Y. (2005). Check 'n' crash: combining static checking and testing. *Proceedings of the IEEE International Conference on Software Engineering (ICSE '05)*, Louis, MO, USA (May 2005), 422–431.

412 Lazaar, N. (2011). CPTEST: a framework for the automatic fault detection, localization and correction of constraint programs. *Proceedings of the Fourth International Conference on Software Testing, Verification, and Validation Workshops*, Berlin, Germany (March 2011), 320–321.

413 Pham, H.V., Lutellier, T., Qi, W., and Tan, L. (2019). CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE '19)*, Montréal, QC, Canada (May 2019), 1027–1038.

414 Brun, Y. and Ernst, M.D. (2004). Finding latent code errors via machine learning over program executions. *Proceedings of the IEEE International Conference on Software Engineering (ICSE '04)*, Edinburgh, UK (May 2004) 480–490.

415 Böhme, M., Soremekun, E.O., Chattopadhyay, S. et al. (2017). Where is the bug and how is it fixed? An experiment with practioners. *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Paderborn, Germany (September 2017), 117–128.

416 Abreu, R., Zoeteweij, P., and Gemund, A.J. (2009). Localizing software faults simultaneously. *Proceedings of the Ninth International Conference on Quality Software*, Jeju, Korea (August 2009), 367–376.

417 Abraham, R. and Erwig, M. (2007). Goaldebug: a spreadsheet debugger for end users. *Proceedings of the IEEE International Conference on Software Engineering (ICSE '07),* Minneapolis, MN, USA (May 2007), 251–260.

418 Gouveia, C., Campos, J., and Abreu, R. (2013). Using HTML5 visualizations in software fault localization. *Proceedings of the first IEEE Working Conference on Software Visualization*, Eindhoven, Netherland (September 2013), 1–10.

419 Machado, P., Campos, J., and Abreu, R. (2013). MZoltar: automatic debugging of android applications. *Proceedings of the International Workshop on Software Development Lifecycle for Mobile*, Saint Petersburg, Russia (August 2013), 9–16.

420 Balogh, G., Horváth, F., and Beszédes, Á. (2019). Poster: aiding java developers with interactive fault localization in eclipse IDE. *Proceedings of the 2019 IEEE International Conference on Software Testing, Verification and Validation*, Xi'an, China (April 2019), 371–374.

421 Ribeiro, H.L., de Araujo, R.P.A., Chaim, M.L. (2018). Jaguar: a spectrum-based fault localization tool for real-world software. *Proceedings of the 2018 IEEE International Conference on Software Testing, Verification and Validation*, Vasteras, Sweden (April 2018), 404–409.

422 Barr, E.T., Marron, M., Maurer, E. et al. (2016). Time-travel debugging for JavaScript/Node.js. *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering,* Seattle, WA, USA (November 2016), 1003–1007.

423 Shu, G., Sun, B., Henderson, T.A.D., and Podgurski, A. (2013). JavaPDG: a new platform for program dependence analysis. *Proceedings of the International Conference on Software Testing, Verification and Validation*, Luxembourg (March 2013), 408–415.

424 Baudry, B., Fleurey, F., and Le Traon, Y. (2006). Improving test suites for efficient fault localization. *Proceedings of the IEEE International Conference on Software Engineering (ICSE '06)*, Shanghai, China (May 2006), 82–91.

425 Flanagan, C. and Freund, S.N. (2010). Adversarial memory for detecting destructive races. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, ON, Canada (June 2010), 244–254.

426 Zhang, C., Su, T., Yan, Y. et al. (2019). Finding and understanding bugs in software model checkers. *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Tallinn, Estonia (August 2019), 763–773.

427 Bagherzadeh, M., Hili, N., and Dingel, J. (2017). Model-level, platform-independent debugging in the context of the model-driven development of real-time systems. *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Paderborn, Germany (September 2017), 419–430.

428 Kellogg, M. (2016). Combining bug detection and teset case generation. *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Seattle, WA, USA (November 2016), 1124–1126.

429 Gyori, A., Lambeth, B., Shi, A. et al. (2016). NonDex: a tool for detecting and debugging wrong assumptions on java API specifications. *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Seattle, WA, USA (November 2016), 993–997.

430 Sorrentino, F., Farzan, A., and Parthasarathy, M. (2010). Penelope: weaving threads to expose atomicity violations. *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, Santa Fe, NM, USA (November 2010), 37–46.

431 Pastore, F., Mariani, L., and Goffi, A. (2013). RADAR: a tool for debugging regression problems in C/C++ software. *Proceedings of the International Conference on Software Engineering (ICSE '13)*, San Francisco, CA, USA (May 2013), 1335–1338.

432 Engler, D. and Ashcraft, K. (2003). Racerx: effective, static detection of race conditions and deadlocks. *Proceedings of the ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, USA (October 2003), 237–252.

433 Salvaneschi, G. and Mezini, M. (2016). Debugging for reactive programming. *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE '16)*, Austin, TX, USA (May 2016), 796–807.

434 Banken, H., Meijer, E., and Gousios, G. (2018). Debugging data flows in reactive programs. *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE '18)*, Gothenburg, Sweden (May 2018), 752–763.

435 Andrews, M. (2003). Signpost: matching program behavior against known faults. *IEEE Software* 20 (6): 84–89.

436 Chowdhury, S.A. (2018). Automatically finding bugs in commercial cyber-physical system development tool chains. *Proceedings of the IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE Companion '18)*, Gothenburg, Sweden (May 2018), 506–508.

437 Guo, X. (2016). SmartDebug: an interactive debug assistant for java. *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Seattle, WA, USA (November 2016), 1127–1129.

438 Pastore, F. and Mariani, L. (2017). VART: a tool for the automatic detection of regression faults. *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Paderborn, Germany (September 2017), 964–968.

439 Hao, D., Zhang, L., Zhang, L. et al. (2009). VIDA: visual interactive debugging. *Proceedings of the International Conference on Software Engineering (ICSE '09)*, Vancouver, BC, Canada (May 2009), 583–586.

440 Servant, F. and Jones, J.A. (2012). WhoseFault: automatic developer-to-fault assignment through fault localization. *Proceedings of the International Conference on Software Engineering (ICSE '12)*, Zurich, Switzerland (June 2012), 36–46.

441 Perez, A., Abreu, R., and d'Amorim, M. (2017). Prevalence of single-fault fixes and its impact on fault localization. *Proceedings of the 2017 IEEE International Conference on Software Testing, Verification and Validation*, Tokyo, Japan (March 2017), 12–22.

442 Ang, A., Perez, A., Deursen, A.V., and Abreu, R. (2017). Revisiting the practical use of automated software fault localization techniques. *Proceedings of the 2017 IEEE 28th International Symposium on Software Reliability Engineering Workshops (ISSREW '17)*, Toulouse, France (23–26 October 2017), 175–182. IEEE. https://doi.org/10.1109/ISSREW.2017.68.

443 Hamill, M. and Goseva-Popstojanova, K. (2009). Common trends in software fault and failure data. *IEEE Transactions on Software Engineering* 35 (4): 484–496.

444 Lucia, L., Thung, F., Lo, D., and Jiang, L. (2012). Are faults localizable? *Proceedings of the IEEE Working Conference on Mining Software Repositories*, Zurich, Switzerland (June 2012), 74–77.

445 Dean, B.C., Pressly, W.B., Malloy, B.A., and Whitley, A.A. (2009). A linear programming approach for automated localization of multiple faults. *Proceedings of the International Conference on Automated Software Engineering*, Auckland, New Zealand (November 2009), 640–644.

446 Dickinson, W., Leon, D., and Fodgurski, A. (2001). Finding failures by cluster analysis of execution profiles. *Proceedings of the International Conference on Software Engineering (ICSE '01)*, Toronto, ON, Canada (May 2001), 339–348.

447 Gong, C., Zheng, Z., Zhang, Y. et al. (2012). Factorising the multiple fault localization problem: adapting single-fault localizer to multi-fault programs. *Proceedings of the Asia-Pacific Software Engineering Conference*, Hong Kong (December 2012), 729–732.

448 Jeffrey, D., Gupta, N., and Gupta, R. (2009). Effective and efficient localization of multiple faults using value replacement. *Proceedings of the International Conference on Software Maintenance*, Edmonton, AB, Canada (September 2009), 221–230.

449 Jones, J.A., Bowring, J., and Harrold, M.J. (2007). Debugging in parallel. *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, London, UK (July 2007), 16–26.

450 Podgurski, A., Leon, D., Francis, P. et al. (2003). Automated support for classifying software failure reports. *Proceedings of the International Conference on Software Engineering (ICSE '03)*, Portland, OR, USA (May 2003), 465–477.

451 Steimann, F. and Bertschler, M. (2009). A simple coverage-based locator for multiple faults. *Proceedings of the International Conference on Software Testing, Verification and Validation*, Denver, CO, USA (April 2009), 366–375.

452 Steimann, F. and Frenkel, M. (2012). Improving coverage-based localization of multiple faults using algorithms from integer linear programming. *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE '12)*, Dallas, TX, USA (November 2012), 121–130.

453 Wei, Z. and Han, B. (2013). Multiple-bug oriented fault localization: a parameter-based combination approach. *Proceedings of the Seventh International Conference on Software Security and Reliability Companion*, Gaithersburg, MD, USA (June 2013), 125–130.

454 Xue, X. and Namin, A.S. (2013). How significant is the effect of fault interactions on coverage-based fault localizations? *Proceedings of the 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, Baltimore, MD, USA (October 2013), 113–122.

455 Zheng, A.X., Jordan, M.I., Liblit, B. et al. (2006). Statistical debugging: simultaneous isolation of multiple bugs. *Proceedings of the International Conference on Machine Learning*, Pittsburgh, PA, USA (June 2006), 26–29.

456 Zakari, A., Lee, S.P., Abreu, R. et al. (2020). Multiple fault localization of software programs: a systematic literature review. *Information and Software Technology* 124: 106312.

457 Zheng, Y., Wang, Z., Fan, X. et al. (2018). Localizing multiple software faults based on evolution algorithm. *Journal of Systems and Software* 139: 107–123.

458 Yan, X., Liu, B., and Li, J. (2017). The failure behaviors of multi-faults programs: an empirical study. *Proceedings of the IEEE International Conference on Software*

Quality, Reliability, and Security Companion, Prague, Czech Republic (July 2017), 1–7.

459 Cardoso, N. and Abreu, R. (2013). A kernel density estimate-based approach to component goodness modeling. *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence,* Bellevue, WA, USA (14–18 July 2013), 152–158.

460 de Kleer, J. (2009). Diagnosing multiple persistent and intermittent faults. *Proceedings of the International Joint Conference on Artificial Intelligence*, Pasadena, CA, USA (July 2009), 733–738.

461 Li, Z., Wu, Y., and Liu, Y. (2019). An empirical study of bug isolation on the effectiveness of multiple fault localization. *Proceedings of the IEEE International Conference on Software Quality, Reliability, and Security (QRS '19)*, Sofia, Bulgaria (July 2019), 18–25.

462 Hammer, C., Dolby, J., Vaziri, M., and Tip, F. (2008). Dynamic detection of atomic-set-serializability violations. *Proceedings of the International Conference on Software Engineering (ICSE '08)*, Leipzig, Germany (May 2008), 231–240.

463 Liu, C., Zhang, X., and Han, J. (2008). A systematic study of failure proximity. *IEEE Transactions on Software Engineering* 34 (6): 826–843.

464 Gao, R. and Wong, W.E. (2019). MSeer-an advanced technique for locating multiple bugs in parallel. *IEEE Transactions on Software Engineering* 45 (3): 301–318.

465 Zhang, X., Zheng, Z., and Cai, K. (2018). Exploring the usefulness of unlabelled test cases in software fault localization. *Journal of Systems and Software* 136: 278–290.

466 Xu, Y., Yin, B., Zheng, Z. et al. (2019). Robustness of spectrum-based fault localisation in environments with labelling perturbations. *Journal of Systems and Software* 147: 172–214.

467 Hierons, R.M. (2012). Oracles for distributed testing. *IEEE Transactions on Software Engineering* 38 (3): 629–641.

468 Hierons, R.M. (2009). Verdict functions in testing with a fault domain or test hypotheses. *ACM Transactions on Software Engineering and Methodology* 18 (4): 1–19.

469 Afshan, S., McMinn, P., and Stevenson, M. (2013). Evolving readable string test inputs using a natural language model to reduce human oracle cost. *Proceedings of the IEEE Sixth International Conference on Software Testing, Verification and Validation*, Luxembourg (March 2013), 352–361.

470 Harman, M., Kim, S.G., Lakhotia, K. et al. (2010). Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. *Proceedings of the International Conference on Software Testing, Verification, and Validation Workshops*, Paris, France (April 2010), 182–191.

471 McMinn, P., Stevenson, M., and Harman, M. (2010). Reducing qualitative human oracle costs associated with automatically generated test data. *Proceedings of the*

International Workshop on Software Test Output Validation, Trento, Italy (July 2010), 1–4.

472 He, Z., Chen, Y., Huang, E. et al. (2019). A system identification based oracle for control-CPS software fault localization. *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE '19)*, Montréal, QC, Canada (May 2019), 116–127.

473 Diaz, E., Tuya, J., and Blanco, R. (2003). Automated software testing using a metaheuristic technique based on tabu search. *Proceedings of the Conference on Automated Software Engineering*, Montreal, QC, Canada (October 2003), 310–313.

474 Artzi, S., Kiezun, A., Dolby, J. et al. (2010). Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Transactions on Software Engineering* 36 (4): 474–494.

475 Artzi, S., Dolby, J., Tip, F., and Pistoia, M. (2010). Directed test generation for effective fault localization. *Proceedings of the IEEE International Symposium on Software Testing and Analysis,* Trento, Italy (July 2010), 49–60.

476 Xu, Z., Liu, P., Zhang, X., and Xu, B. (2016). Python predictive analysis for bug detection. *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Seattle, WA, USA (November 2016), 121–132.

477 Jiang, B., Chan, W.K., and Tse, T.H. (2011). On practical adequate test suites for integrated test case prioritization and fault localization. *Proceedings of the International Conference on Quality Software*, Madrid, Spain (13–14 July 2011), 21–30.

478 Jiang, B., Zhai, K., Chan, W.K. et al. (2013). On the adoption of MC/DC and control-flow adequacy for a tight integration of program testing and statistical fault localization. *Information and Software Technology* 55 (5): 897–917.

479 Santelices, R., Jones, J.A., Yu, Y., and Harrold, M.J. (2009). Lightweight fault-localization using multiple coverage types. *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering (ICSE '09)*, Vancouver, BC, Canada (16–24 May 2009), 56–66. IEEE. https://doi.org/10.1109/ICSE.2009.5070508.

480 Bandyopadhyay, A. (2011). Improving spectrum-based fault localization using proximity-based weighting of test cases. *Proceedings of the IEEE International Symposium on Automated Software Engineering*, Lawrence, KS, USA (November 2011), 660–664.

481 Bandyopadhyay, A. and Ghosh, S. (2011). Proximity based weighting of test cases to improve spectrum based fault localization. *Proceedings of the IEEE International Symposium on Automated Software Engineering*, Lawrence, KS, USA (November 2011), 420–423.

482 Cai, K., Jing, T., and Bai, C. (2005). Partition testing with dynamic partitioning. *Proceedings of the Annual International Computer Software and Applications Conference (COMPSAC '05)*, Edinburgh, UK (July 2005), 113–116.

483 Campos, J., Abreu, R., Fraser, G., and Amorim, M. (2013). Entropy-based test generation for improved fault localization. *Proceedings of the 28th International*

Conference on Automated Software Engineering, Silicon Valley, CA, USA (November 2013), 257–267.

484 Gong, L., Lo, D., Jiang, L., and Zhang,H. (2012). Diversity maximization speedup for fault. *Proceedings of the International Conference on Automated Software Engineering*, Essen, Germany (September 2012), 30–39.

485 Gonzalez-Sanchez, A., Piel, E., Gross, H., and van Gemund, A.J. (2010). Prioritizing tests for software fault localization. *Proceedings of the International Conference on Quality Software*, Zhangjiajie, China (July 2010), 42–51.

486 Gonzalez-Sanchez, A., Abreu, R., Gross, H., and van Gemund, A.J.C. (2011). An empirical study on the usage of testability information to fault localization in software. *Proceedings of the ACM Symposium on Applied Computing*, TaiChung, Taiwan (March 2011), 1398–1403.

487 Gonzalez-Sanchez, A., Abreu, R., Gross, H., and van Gemund, A.J.C. (2011). Prioritizing tests for fault localization through ambiguity group reduction. *Proceedings of the International Conference on Automated Software Engineering*, Lawrence, KS, USA (November 2011), 83–92.

488 Hao, D., Zhang, L., Zhong, H. et al. (2005). Eliminating harmful redundancy for testing-based fault localization using test suite reduction: an experimental study. *Proceedings of the International Conference on Software Maintenance,* Budapest, Hungary (September 2005), 683–686.

489 Hui, Z. (2016). Fault localization method generated by regression test cases on the basis of genetic immune algorithm. *Proceedings of the IEEE 40th Annual Computer Software and Applications Conference Workshops (COMPSAC Workshops '16)*, Atlanta, GA, USA (June 2016), 46–51.

490 Jiang, B. and Chan, W.K. (2010). On the integration of test adequacy, test case prioritization, and statistical fault localization. *Proceedings of the International Conference on Quality Software*, Zhangjiajie, China (July 2010), 377–384.

491 Kim, J., Park, J., and Lee, E. (2016). A new spectrum-based fault localization with the technique of test case optimization. *Journal of Information Science and Engineering* 32 (1): 177–196.

492 Li, Y., Chen, J., Ni, F. et al. (2015). Selecting test cases for result inspection to support effective fault localization. *Journal of Computer Science and Engineering* 9 (3): 142–154.

493 Vidacs, L., Bezedes, A., Tengeri, D. et al. (2014). Test suite reduction for fault detection and localization: a combined approach. *Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering*, Antwerp, Belgium (February 2014), 204–213.

494 Weiglhofer, M., Fraser, G., and Wotawa, F. (2009). Using spectrum-based fault localization for test case grouping. *Proceedings of the International Conference on Automated Software Engineering*, Auckland, New Zealand (November 2009), 630–634.

495 Yu, Y., Jones, J., and Harrold, M.J. (2008). An empirical study of the effects of test-suite reduction on fault localization. *Proceedings of the International Conference on Software Engineering (ICSE '08)*, Leipzig, Germany (May 2008), 201–210.

496 Zhang, X., Wang, Z., Zhang, W. et al. (2015). Spectrum-based fault localization method with test case reduction. *Proceedings of the IEEE 39th Annual Computer Software and Applications Conference Workshops (COMPSAC Workshops '15)*, Taichung, Taiwan (July 2015), 548–549.

497 Naish, L., Lee, H.J., and Ramamohanarao, K. (2009). Spectral debugging with weights and incremental ranking. *Proceedings of the Asia-Pacific Software Engineering Conference*, Batu Ferringhi, Malaysia (December 2009), 168–175.

498 Gao, Y., Zhang, Z., Zhang, L. et al. (2013). A theoretical study: the impact of cloning failed test cases on the effectiveness of fault localization. *Proceedings of the International Conference on Quality Software*, Nanjing, China (July 2013), 288–291.

499 Zhang, L., Yan, L., Zhang, Z. et al. (2017). A theoretical analysis on cloning the failed test cases to improve spectrum-based fault localization. *Journal of Systems and Software* 129: 35–57.

500 Röβler, J., Fraser, G., Zeller, A., and Orso, A. (2012). Isolating failure causes through test case generation. *Proceedings of the International Symposium on Software Testing and Analysis*, Minneapolis, MN, USA (July 2012), 309–319.

501 Li, X., Wong, W.E., Gao, R. et al. (2018). Genetic algorithm-based test generation for software product line with the integration of fault localization techniques. *Empirical Software Engineering* 23: 1–51.

502 Liu, M., Liu, P., Yang, X., and Zhao, L. (2016). Fault localization guided execution comparison for failure comprehension. *Proceedings of the IEEE International Conference on Software Quality, Reliability, and Security Companion (QRS-C '16)*, Vienna, Austria (August 2016), 163–166.

503 Chen, R., Chen, S., and Zhang, N. (2016). Iterative path clustering for software fault localization. *Proceedings of the IEEE International Conference on Software Quality, Reliability, and Security Companion (QRS-C '16)*, Vienna, Austria (August 2016), 292–297.

504 Lekivetz, R. and Morgan, J. (2018). Fault localization: analyzing covering arrays given prior information. *Proceedings of the IEEE International Conference on Software Quality, Reliability, and Security Companion*, Lisbon, Portugal (July 2018), 116–121.

505 Perez, A., Abreu, R., and van Deursen, A. (2017). A test-suite diagnosability metric for spectrum-based fault localization approaches. *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE '17)*, Buenos Aires, Argentina (May 2017), 654–664.

506 Just, R., Parnin, C., Drosos, I., and Ernst, M.D. (2018). Comparing developer-provided to user-provided tests for fault localization and automated program

repair. *Proceedings of the 2018 ACM International Symposium on Software Testing and Analysis*, Amsterdam, Netherlands (July 2018), 287–297.

507 Gambi, A., Bell, J., and Zeller, A. (2018). Practical test dependency detection. *Proceedings of the 2018 IEEE International Conference on Software Testing, Verification and Validation*, Vasteras, Sweden (April 2018), 1–11.

508 Budd, T.A. and Angluin, D. (1982). Two notions of correctness and their relation to testing. *Acta Infomatica* 18 (1): 31–45.

509 Bandyopadhyay, A. (2012). Mitigating the effect of coincidental correctness in spectrum based fault localization. *Proceedings of the International Conference on Software Testing, Verification, and Validation*, Montreal, QC, Canada (April 2012), 479–482.

510 Hierons, R.M. (2006). Avoiding coincidental correctness in boundary value analysis. *ACM Transactions on Software Engineering and Methodology* 15 (3): 227–241.

511 Li, Y. and Liu, C. (2012). Using cluster analysis to identify coincidental correctness in fault localization. *Proceedings of the International Conference on Computational and Information Sciences*, Chongqing, China (August 2012), 357–360.

512 Masri, W. and Assi, R.A. (2010). Cleansing test suites from coincidental correctness to enhance fault-localization. *Proceedings of the International Conference on Software Testing, Verification, and Validation*, Paris, France (April 2010), 165–174.

513 Miao, Y., Chen, Z., Li, S. et al. (2012). Identifying coincidental correctness for fault localization clustering test cases. *Proceedings of the 24th International Conference on Software Engineering and Knowledge Engineering*, Redwood City, CA, USA (1–3 July 2012), 267–272. KSI Research Inc.

514 Wang, X., Cheung, S.C., Chan, W.K., and Zhang, Z. (2009). Taming coincidental correctness: refine code coverage with context pattern to improve fault localization. *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, Vancouver, BC, Canada (16–24 May 2009), 45–55. IEEE. https://doi.org/10.1109/ICSE.2009.5070507.

515 Zhang, Z., Chan, W.K., and Tse, T.H. (2012). Fault localization based only on failed runs. *Computer* 45 (6): 64–71.

516 Zhou, X., Wang, H., and Zhao, J. (2015). A fault-localization approach based on the coincidental correctness probability. *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security*, Vancouver, BC, Canada (August 2015), 292–297.

517 Liu, Y., Li, M., Wu, Y., and Li, Z. (2019). A weighted fuzzy classification approach to identify and manipulate coincidental correct test cases for fault localization. *Journal of Systems and Software* 151: 20–37.

518 Masri, W. and Assi, R.A. (2014). Prevalence of coincidental correctness and mitigation of its impact on fault localization. *ACM Transactions on Software Engineering and Methodology* 23 (1): 1–28.

519 Zhang, Z., Jiang, B., Chan, W.K., and Tse, T.H. (2011). Precise propagation of fault-failure correlations in program flow graphs. *Proceedings of the 35th Annual International Computer Software and Applications Conference (COMPSAC '11)*, Munich, Germany (18–22 July 2011), 58–67.

520 Hofer, B. (2017). Removing coincidental correctness in spectrum-based fault localization for circuit and spreadsheet debugging. *Proceedings of the 2017 IEEE 28th International Symposium on Software Reliability Engineering Workshops (ISSREW '17)*, Toulouse, France (23–26 October 2017), 199–206. IEEE. https://doi.org/10.1109/ISSREW.2017.18.

521 Debroy, V. and Wong, W.E. (2011). On the consensus-based application of fault localization techniques. *Proceedings of the IEEE Annual Computer Software and Applications Conference Workshops (COMPSAC Workshops '11)*, Munich, Germany (July 2011), 506–511.

522 Lucia, L., Lo, D., and Xia, X. (2014). Fusion fault localizers. *Proceedings of the IEEE International Conference on Automated Software Engineering*, Vasteras, Sweden (September 2014), 127–138.

523 Tang, C.M., Keung, J., Yu, Y.T., and Chan, W.K. (2016). DFL: dual-service fault localization. *Proceedings of the IEEE International Conference on Software Quality, Reliability, and Security*, Vienna, Austria (August 2016), 412–422.

524 Wang, S., David, L., Jiang, L. (2011). Search-based fault localization. *Proceedings of the 2011 IEEE/ACM International Conference on Automated Software Engineering*, Lawrence, KS, USA (November 2011), 556–559.

525 Leitao-Junior, P.S., Freitas, D.M., Vergilio, S.R. et al. (2020). Search-based fault localisation: a systematic mapping study. *Information and Software Technology* 123: 106295.

526 Lei, Y., Mao, X., Dai, Z., and Wang, C. (2012). Effective statistical fault localization using program slices. *Proceedings of the Annual IEEE International Computer Software and Applications Conference (COMPSAC '12)*, Izmir, Turkey (July 2012), 1–10.

527 Wen, W. (2012). Software fault localization based on program slicing Spectrum. *Proceedings of the International Conference on Software Engineering (ICSE '12)*, Zurich, Switzerland (June 2012), 1511–1514.

528 Reis, S., Abreu, R., and d'Amorim, M. (2019). Demystifying the combination of dynamic slicing and spectrum-based fault localization. *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, Macao, China (10–16 August 2019), 4760–4766. https://doi.org/10.24963/ijcai.2019/661.

529 Li, X. and Orso, A. (2020). More accurate dynamic slicing for better supporting software debugging. *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, Porto, Portugal (October 2020). IEEE.

530 Christi, A., Olson, M.L., Alipour, M.A., and Groce, A. (2018). Reduce before you localize: delta-debugging and spectrum-based fault localization. *Proceedings of the 2018 IEEE 29th International Symposium on Software Reliability Engineering*

Workshops (ISSREW '18), Memphis, TN, USA (15–18 October 2018), 184–191. IEEE. https://doi.org/10.1109/ISSREW.2018.00005.

531 Xuan, J. and Morperrus, M. (2014). Learning to combine multiple ranking metrics for fault localization. *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, Victoria, BC, Canada (September 2014), 191–200.

532 Gopinath, D., Zaeem, R.N., and Khurshid, S. (2012). Improving the effectiveness of spectra-based fault localization using specifications. *Proceedings of the International Conference on Automated Software Engineering*, Essen, Germany (September 2012), 40–49.

533 Burger, M. and Zeller, A. (2011). Minimizing reproduction of software failures. *Proceedings of the International Symposium on Software Testing and Analysis*, Toronto, BC, Canada (July 2011), 221–231.

534 Artho, C., Havelund, K., and Biere, A. (2003). High-level data races. *Journal on Software Testing, Verification and Reliability* 13 (4): 207–227.

535 Savage, S., Burrows, M., Nelson, G. et al. (1997). Eraser: a dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems* 15 (4): 391–411.

536 Farzan, A. and Madhusudan, P. (2006). Causal atomicity. *Proceedings of the International Conference Computer Aided Verification*, Seattle, WA, USA (August 2006), 315–328.

537 Flanagan, C. and Freund, S.N. (2004). Atomizer: a dynamic atomicity checker for multithreaded programs. *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Venice, Italy (January 2004), 256–267.

538 Flanagan, C., Freund, S.N., and Qadeer, S. (2005). Exploiting purity for atomicity. *IEEE Transactions on Software Engineering* 31 (4): 275–291.

539 Farzan, A., Madhusudan, P., and Sorrentino, F. (2009). Meta-analysis for atomicity violations under nested locking. *Proceedings of the International Conference Computer Aided Verification*, Grenoble, France (July 2009), 248–262

540 Sen, K., Rosu, G., and Agha, G. (2003). Runtime safety analysis of multithreaded programs. *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, Helsinki, Finland (September 2003), 337–346.

541 Wang, L. and Stoller, S.D. (2006). Accurate and efficient runtime detection of atomicity errors in concurrent programs. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA (29–31 March 2006), 137–146. ACM. https://doi.org/10.1145/1122971.1122993.

542 Xu, M., Bodik, R., and Hill, M.D. (2005). A serializability violation detector for shared-memory server programs. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, IL, USA (12–15 June 2005), 1–14.

543 Burckhardt, S., Alur, R., and Martin, M.M.K. (2007). Checkfence: checking consistency of concurrent data types on relaxed memory models. *Proceedings of*

the ACM SIGPLAN Conference on Programming Language Design and Implementation, San Diego, CA, USA (10–13 June 2007), 12–21.

544 Kidd, N., Reps, T., Dolby, J., and Vaziri, M. (2009). Finding concurrency-related bugs using random isolation. *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation*, Savannah, GA, USA (18–20 January 2009), 198–213.

545 Musuvathi, M., Qadeer, S., Ball, T. et al. (2008). Finding and reproducing heisenbugs in concurrent programs. *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, USA (December 2008), 267–280.

546 Shacham, O., Sagiv, M., and Schuster A. (2005). Scaling model checking of dataraces using dynamic information. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Chicago, IL, USA (June 2005), 107–118.

547 Park, S., Lu, S., and Zhou, Y. (2009). Ctrigger: exposing atomicity violation bugs from their hiding places. *Proceedings of the International Conference on Architectural Support for Programming Language*, Washington, DC, USA (7–11 March 2009), 25–36.

548 Park, S. (2013). Debugging non-deadlock concurrency bugs. *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, Lugano, Switzerland (July 2013), 358–361.

549 Wang, W., Wu, C., Yew, P. (2014). Concurrency bug localization using shared memory access pairs. *Proceedings of the 19th ACM SIGPLAN symposium on Principles and Practices of Parallel Programming*, Orlando, FL, USA (February 2014), 375–376.

550 Torlak, E., Vaziri, M., and Dolby, J. (2010). MemSAT: checking axiomatic specifications of memory models. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, BC, Canada (June 2010), 341–350.

551 Koca, F., Sozer, H., and Abreu, R. (2013). Spectrum-based fault localization for diagnosing concurrency faults. *Proceedings of the International Conference on Testing Software and Systems*, Istanbul, Turkey (November 2013), 239–254.

552 Xu, J., Lei, Y., and Carver, R. (2017). Using delta debugging to minimize stress tests for concurrent data structures. *Proceedings of the 2017 IEEE International Conference on Software Testing, Verification and Validation*, Tokyo, Japan (March 2017), 35–46.

553 Panko, R.R. and Ordway, N. (2008). Sarbanes-oxley: what about all the spreadsheets? arXiv preprint arXiv:0804.0797. https://arxiv.org/ftp/arxiv/papers/0804/0804.0797.pdf.

554 Hermans, F., Pinzger, M., and van Deursen, A. (2011). Supporting professional spreadsheet users by generating leveled dataflow diagrams. *Proceedings of*

the International Conference on Software Engineering (ICSE '11), Waikiki, HI, USA (May 2011), 451–460.

555 Panko, R. (2006). Facing the problem of spreadsheet errors. *Decision Line* 37 (5): 8–10.

556 Panko, R.R. (2000). Spreadsheet errors: what we know. What we think we can do. arXiv preprint arXiv:0802.3457.

557 Reinhart, C.M. and Rogoff, K.S. (2010). Growth in a time of debt. *American Economic Review* 100 (2): 573–578.

558 Abraham, R. and Erwig, M. (2004). Header and unit inference for spreadsheets through spatial analyses. *Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing*, Roma, Italy (September 2004), 165–172.

559 Abraham, R. and Erwig, M. (2007). UCheck: a spreadsheet type checker for end users. *Journal of Visual Languages and Computing* 18 (1): 71–95.

560 Ahmad, Y., Antoniu, T., Goldwater, S., and Krishnamurthi, S. (2003). A type system for statically detecting spreadsheet errors. *Proceedings of the IEEE International Symposium on Automated Software Engineering*, Montreal, QC, Canada (October 2003), 174–183.

561 Bregar, A. (2004). Complexity metrics for spreadsheet models. arXiv preprint arXiv:0802.3895.

562 Hermans, F., Pinzger, M., and van Deursen, A. (2012). Detecting and visualizing inter-worksheet smells in spreadsheets. *Proceedings of the International Conference on Software Engineering (ICSE '12)*, Zurich, Switzerland (June 2012), 441–451.

563 Hermans, F., Sedee, B., Pinzger, M., and van Deursen, A. (2013). Data clone detection and visualization in spreadsheets. *Proceedings of the 2013 35th International Conference on Software Engineering (ICSE '13)*, San Francisco, CA, USA (18–26 May 2013), 292–301. IEEE. https://doi.org/10.1109/ICSE.2013.6606575.

564 Hodnigg, K. and Mittermeir, R.T. (2008). Metrics-based spreadsheet visualization: support for focused maintenance. arXiv preprint arXiv:0809.3009.

565 Hofer, B., Riboira, A., Wotawa, F. et al. (2013). On the empirical evaluation of fault localization techniques for spreadsheets. *International Conference on Fundamental Approaches to Software Engineering*, Rome, Italy (March 2013), 68–82.

566 Rothermel, K.J., Li, L., DuParis, C., and Burnett, M. (1998). What you see is what you test: a methodology for testing form-based visual programs. *Proceedings of the 1998 International Conference on Software Engineering (ICSE '98)*, Kyoto, Japan (April 1998), 198–207.

567 Ruthruff, J., Creswick, E., Burnett, M. et al. (2003). End-user software visualizations for fault localization. *Proceedings of the ACM Symposium on Software Visualization*, San Diego, CA, USA (June 2003), 123–132.

568 Abreu, R., Cunha, J., Fernandes, J.P. et al. (2014). Faultysheet detective: when smells meet fault localization. *Proceedings of the 2014 IEEE International*

Conference on Software Maintenance and Evolution, Victoria, BC, Canada (28 September to 3 October 2014), 625–628. IEEE.

569 Abreu, R., Cunha, J., Fernandes, J.P. et al. (2014). Smelling faults in spreadsheets. *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, Victoria, BC, Canada (28 September to 3 October 2014), 111–120. IEEE.

570 Abreu, R., Hofer, B., Perez, A., and Wotawa, F. (2014). Using constraints to diagnose faulty spreadsheets. *Software Quality Journal* 23 (2): 297–322.

571 Abreu, R., Riboira, A., and Wotawa, F. (2012). Constraint-based debugging of spreadsheets. *Proceedings of the 15th Iberoamerican Conference on Software Engineering (CIbSE 2012)*, Buenos Aires, Argentina (24–27 April 2012), 1–14.

572 Getzner, E., Hofer, B., and Wotawa, F. (2017). Improving spectrum-based fault localization for spreadsheet debugging. *Proceedings of the IEEE International Conference on Software Quality, Reliability, and Security*, Prague, Czech Republic (July 2017), 102–113.

573 Almasi, M.M., Hemmati, H., and Fraser, G. (2018). Search-based detection of deviation failures in the migration of legacy spreadsheet applications. *Proceedings of the 2018 ACM International Symposium on Software Testing and Analysis*, Amsterdam, Netherlands (July 2018), 266–275.

574 Hofer, B., Perez, A., Abreu, R., and Wotawa, F. (2015). On the empirical evaluation of similarity coefficients for spreadsheets fault localization. *Automated Software Engineering* 22 (1): 47–74.

575 Hofer, B. and Wotawa, F. (2015). Fault localization in the light of faulty user input. *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security*, Vancouver, BC, Canada (August 2015), 282–291.

576 Panko, R.R. (1999). Applying code inspection to spreadsheet testing. *Journal of Management Information Systems* 16 (2): 159–176.

577 Badame, S., Dig, D. (2012). Refactoring meets spreadsheet formulas. *Proceedings of the International Conference on Software Maintenance*, Trento, Italy (September 2012), 399–409.

578 Cunha, J., Erwig, M., Saraiva, J. (2010). Automatically inferring classsheet models from spreadsheets. *Proceedings of the IEEE Symposium on Visual Language and Human-Centric Computing*, Leganes-Madrid, Spain (September 2010), 93–100.

579 Cunha, J., Fernandes, J.P., Mendes, J., and Saraiva, J. (2012). MDSheet: a framework for model-driven Spreadsheet Engineering. *Proceedings of the IEEE International Conference on Software Engineering (ICSE '12)*, Zurich, Switzerland (June 2012), 1395–1398.

580 Lee, H.J., Naish, L., and Ramamohanarao, K. (2009). Study of the relationship of bug consistency with respect to performance of spectra metrics. *Proceedings of the 2009 2nd IEEE International Conference on Computer Science and Information*

Technology, Beijing, China (8–11 August 2009), 501–508. IEEE. https://doi.org/10.1109/ICCSIT.2009.5234512.

581 Xie, X., Chen, T.Y., Kuo, F.-C., and Xu, B.W. (2013). A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology* 22 (4): 1–40.

582 Xie, X., Kuo, F., Chen, T.Y. et al. (2013). Provably optimal and human-competitive results in SBSE for spectrum based fault localization. *Proceedings of the International Symposium on Search Based Software Engineering*, Saint Petersburg, Russia (August 2013), 224–238.

583 Yoo, S., Xie, X., Kuo, F. et al. Human competitiveness of genetic programming in spectrum-based fault localisation: theoretical and empirical analysis. *ACM Transactions on Software Engineering and Methodology* 26 (1): 1–30, Article 4.

584 Ju, X., Chen, X., Yang, Y. et al. (2017). An in-depth study of the efficiency of risk evaluation formulas for multi-fault localization. *Proceedings of the IEEE International Conference on Software Quality, Reliability, and Security Companion*, Prague, Czech Republic (July 2017), 304–310.

585 Naish, L., Lee, H.J., and Ramamohanarao, K. (2012). Spectral debugging: how much better can we do. *Proceedings of the Australian Software Engineering Conference*, Melbourne, Australia (January 2012), 96–106.

586 Naish, L. and Lee, H.J. (2013). Duals in spectral fault localization. *Proceedings of the Australian Software Engineering Conference*, Melbourne, Australia (June 2013), 51–59.

587 Chen, T.Y., Xie, X., Kuo, F., and Xu, B. (2015). A revisit of a theoretical analysis on spectrum-based fault localization. *Proceedings of the IEEE 39th Annual Computer Software and Applications Conference (COMPSAC '15)*, Taichung, Taiwan (July 2015), 17–22.