Revisiting the Practical Use of Automated Software Fault Localization Techniques

Aaron Ang*§, Alexandre Perez†, Arie van Deursen*, Rui Abreu‡
*Delft University of Technology, The Netherlands
†University of Porto, Portugal
‡University of Lisbon, Portugal
§Palo Alto Research Center, USA
a.w.z.ang@student.tudelft.nl, alexandre.perez@fe.up.pt, arie.vandeursen@tudelft.nl, rui@computer.org

***Abstract*—In the last two decades, a great amount of effort has been put in researching automated debugging techniques to support developers in the debugging process. However, in a widely cited user study published in 2011, Parnin and Orso found that research in automated debugging techniques made assumptions that do not hold in practice, and suggested four research directions to remedy this: absolute evaluation metrics, result comprehension, ecosystems, and user studies.**

In this study, we revisit the research directions proposed by the authors, offering an overview of the progress that the research community has made in addressing them since 2011. We observe that new absolute evaluation metrics and result comprehension techniques have been proposed, while research in ecosystems and user studies remains mostly unexplored. We analyze what is hard about these unexplored directions and propose avenues for further research in the area of fault localization.

***Index Terms*—Software Fault Localization; Debugging; Literature Survey.**

I. Introduction

Software systems are complex and error-prone, likely to expose failures to the end user. When a failure occurs, the developer has to debug the system to eliminate the failure. This debugging process can be described in three phases: fault localization, fault understanding, and fault correction [1]. This process is time-consuming and can account for 30% to 90% of the software development cycle [2]–[4].

Traditionally, developers use four different approaches to debug a software system, namely program logging, assertions, breakpoints and profiling [5]. These techniques provide an intuitive approach to localize the root cause of a failure, but, as one might expect, are less effective in the massive size and scale of software systems today.

Therefore, in the last decades a lot of research has been performed on improving and developing *advanced* fault localization techniques [5] such that they are applicable to the software systems of today. Specifically, the most prominent techniques are spectrum-based fault localization (SBFL) techniques. SBFL techniques pinpoint faults in code based on execution information of a program, also known as a program spectrum [6]. It does this by outputting a list of suspicious components, for example statements or methods, ranked by their suspiciousness. Intuitively, if a statement is executed primarily during failed executions, then this statement might be assigned a higher suspiciousness score. Conversely, if a statement is executed primarily during successful executions, then this statement might be assigned a lower suspiciousness score.

While advanced fault localization techniques have proven to be able to pinpoint faults in code, many studies have ignored their practical effectiveness [7]. This issue was raised in 2011 in a study by Parnin and Orso [1], in which they perform a preliminary user study and show evidence that many assumptions made by advanced fault localization techniques do not hold in practice. For example, many studies adopt a metric that is relative to the size of the codebase to evaluate the performance of a debugging technique. If a faulty statement is assigned a rank of 83, while the total lines of code amounts to 4408, then the evaluation metric suggests that the developer has to inspect 1.8% of the codebase, which appears as a positive result. However, Parnin and Orso observed in their user experiment that developers were not able to translate the results into a successful debugging activity [1].

In this paper, we seek to understand the response of the software fault localization (SFL) research community with regard to Parnin and Orso's pioneering study, in which multiple directions are proposed for future research in the area of fault localization. To that end, we conduct a literature survey analyzing papers that build upon Parnin and Orso's study. We assess the progress that has been made since the original study appeared, identify areas that are still open, and give recommendations for future research regarding the practical use of fault localization.

II. Background

To set the scene for our study, we first provide an overview of the four most studied software fault localization techniques and identify existing surveys on such techniques.

Today's most important fault localization techniques can be grouped into four categories: slice-based, spectrum-based, model-based, and information retrieval-based techniques. The first three techniques are discussed because most research has been performed on them compared to other techniques [5]. We discuss information retrieval-based fault localization techniques because they are inherently designed to work on natural languages, which can be useful in providing more context to developers when using SFL techniques in practice.

A. Slice-based Techniques

Static slicing was first introduced by Weiser [8], where irrelevant components of a program are removed from the original set of components to obtain a reduced executable form. This creates a smaller search domain for the developer to locate a fault.

Due to the fact that static slices include every statement that can possibly affect the variables of interest, a constructed slice may still contain statements that are not useful for locating a bug. To deal with this problem, Korel and Laski proposed *dynamic* program slicing [9]. In dynamic slicing, a slice is constructed based on the execution information of a program for a specific input.

B. Spectrum-based Techniques

A spectrum was first introduced by Reps *et al.* [6]. A program spectrum consists of execution information from a perspective of interest. For example, a path spectrum may contain simple information such as whether a path has been executed, also known as the hit spectrum. This kind of information was used to tackle the Y2K problem by Reps *et al.* [6] by comparing multiple path spectra to identify paths that are likely date-dependent.

With this in mind, Collofello and Cousins [10] performed one of the first studies where multiple path spectra are used to localize faults in code. Collofello and Cousins proposed a theory, called relational path analysis, which requires a database that stores correctly executed paths according to test cases that pass successfully. Then, by contrasting a failing execution with the database, execution paths can be pinpointed that are likely to contain the fault.

Collofello and Cousins' work formed the basis for hit spectrum-based fault localization. To formalize their idea, we define the finite set $\mathcal{C} = \langle c_1, c_2, \ldots, c_M \rangle$ of M system components, and the finite set $\mathcal{T} = \langle t_1, t_2, \ldots, t_N \rangle$ of N system transactions, such as test executions. The outcomes of all system transactions are defined as an error vector $e = \langle e_1, e_2, \ldots, e_N \rangle$, where $e_i = 1$ indicates that transaction t_i has failed and $e_i = 0$ otherwise. To keep track of which system components were executed during which system transactions, we construct a $N \times M$ activity matrix $\mathcal{A}$, where $\mathcal{A}_{ij} = 1$ indicates that component c_j was hit during transaction t_i. Given these definitions, SBFL techniques compute statistics such that the suspiciousness score of a system component can be computed.

A popular SBFL technique to compute the suspiciousness score of each system component is Tarantula, proposed by Jones *et al.* [11]. Tarantula was developed to visualize fault localization results based on suspiciousness scores to improve the developer's ability to locate faults.

C. Model-based Techniques

Model-based software fault localization is an application of model-based diagnosis (MBD). MBD was first introduced by Davis [12] and was primarily intended for fault diagnosis in hardware, such as faulty gates in electrical circuits. Subsequently, various studies [13], [14] have refined this area. The underlying theory assumes that there exists a model that defines the correct behavior of a system. Faults are diagnosed when the actual observed behavior differs from the specified behavior.

In 1999, Mateis *et al.* [15] performed the first study where MBD is applied to Java, an imperative programming language. As opposed to models for physical systems, software programs written in an imperative language seldom come with a complete and up-to-date behavioral model. Therefore, for software systems, the model is generated from source code based on the semantics of the programming language. However, this model can be faulty as the source code is likely to contain bugs. Hence, expected results from a test case and its execution are used together with the generated model to diagnose bugs [16].

D. Information Retrieval-based Techniques

Information retrieval (IR) has been most apparent in web search engines but has recently been applied to SFL. The purpose of IR is to retrieve *relevant* documents given a query [17]. In IR-based SFL (IRBSFL), bug reports are used as a search query and source code represents the document collection. To retrieve relevant documents, IRBSFL techniques make use of retrieval models, that essentially return documents that are most similar to the search query. Specifically, retrieval models define how documents and queries are characterized such that, ultimately, the representation of a document and query can be compared to find the most relevant documents. The five generic retrieval models that are used to perform SFL are [18]: Vector Space Model (VSM) [19] , Smoothed Unigram Model (SUM) [18], Latent Dirichlet Allocation (LDA) [20], [21], Latent Semantic Indexing (LSI) [22], [23], Cluster Based Document Model (CBDM) [18].

E. Surveys on Software Fault Localization

Several literature surveys [5], [24] have been performed to help the community get a better understanding of all advances made in SFL.

Recently, Wong *et al.* [5] published a comprehensive literature survey where the body of literature comprises studies published from 1977 to November 2014. The fault localization techniques are categorized into eight groups, namely slice-based, spectrum-based, statistics-based, program state-based, machine learning-based, data mining-based, model-based and miscellaneous techniques. Further, Wong *et al.* discussed several metrics and fault localization tools that are proposed since 1977 and concluded their survey by addressing nine critical aspects in fault diagnosis. This work differs from Wong *et al.* in that we mainly focus on the improvements in SFL regarding its practical issues.

Souza *et al.* [24] presented a fault localization survey, where they addressed the shortcomings of current SBFL techniques to be applied in industry. The authors do this by addressing five aspects of fault localization: techniques, faults, benchmarks, testing information, and practical use. Although Souza *et al.*

focused on the practicality of SBFL, which is similar to this survey, we also survey studies that propose SFL ecosystems.

III. Parnin and Orso's Study

In this section, we first highlight the essence of Parnin and Orso's study [1]: "Are Automated Debugging Techniques Actually Helping Programmers?". Then, we generalize Parnin and Orso's research directions.

A. Summary

Parnin and Orso performed a preliminary user study to examine the usefulness of a popular automated debugging technique in practice to gain insight on how to build better debugging tools. An additional goal was to identify promising research directions in this area.

The authors defined the following hypotheses and research questions.

- *Hypothesis 1:* Programmers who debug with the assistance of automated debugging tools will locate bugs faster than programmers who debug code completely by hand.
- *Hypothesis 2:* The effectiveness of an automated tool increases with the level of difficulty of the debugging task.
- *Hypothesis 3:* The effectiveness of debugging when using a ranking based automated tool is affected by the rank of the faulty statement.
- *Research Question 1:* How do developers navigate a list of statements ranked by suspiciousness? Do they visit them in order of suspiciousness or go from one statement to the other by following a different order?
- *Research Question 2:* Does perfect bug understanding exist? How much effort is actually involved in inspecting and assessing potentially faulty statements?
- *Research Question 3:* What are the challenges involved in using automated debugging tools? What issues or barriers prevent their effective use? Can unexpected, emerging strategies be observed?

Their experiments involved 34 developers divided into four experimental groups: A, B, C, and D. Each participant was assigned two debugging tasks — debug a failure in Tetris (easy) and NanoXML (difficult) — and each group had to use Tarantula [11] for one of the tasks or both. During the experiment, the authors recorded a log of the navigation history of the participants that used Tarantula and made use of a questionnaire in which participants were asked to share their experience and issues.

In the analysis of the results, Parnin and Orso categorized the participants as low, medium, or high performer. The average completion time of the high performers in group A is significantly shorter than the average completion time of the high performers in group B for Tetris, and thus Hypothesis 1 is supported but limited to experts and simpler code. For Hypothesis 2 and Hypothesis 3 no support was found.

Based on the recorded logs and questionnaires, the authors found that developers do not linearly traverse the ranked list, produced by Tarantula. Instead, the participants exhibited some form of jumping between ranked statements, searched for statements in the list to confirm their intuition, or skipped statements that did not appear relevant. In addition, the recorded logs showed evidence that *perfect bug understanding* is not a realistic assumption. On average, developers spent ten additional minutes on searching the diagnosis report after the first encounter with the faulty statement. Regarding Research Question 3, the participants indicated that they prefer more context, e.g. runtime values, or different ways of interacting with the data.

Besides the hypotheses and research questions, Parnin and Orso made several observations and derived research implications as follows.

- *Observation 1:* An automated debugging tool may help ensure developers correct faults instead of simply patching failures.
- *Observation 2:* Providing overviews that cluster results and explanations that include data values, test case information, and information about slices could make faults easier to identify and tools ultimately more effective.
- *Implication 1:* Techniques should focus on improving absolute rank rather than percentage rank.
- *Implication 2:* Debugging tools may be more successful if they focused on searching through or automatically highlighting certain suspicious statements.
- *Implication 3:* Research should focus on providing an ecosystem that supports the entire tool chain for fault localization, including managing and orchestrating test cases.

B. Generalization

The first implication states that future research should improve absolute rank instead of percentage rank. Percentage rank is used in many studies to evaluate the performance of the fault localization technique. However, percentage rank does not scale with the size of a codebase. For example, when a faulty statement is ranked on the 83$^{\text{rd}}$ position as a result of the fault localization technique and the codebase consists of 8300 lines of code, the percentage rank is $\frac{83}{8300} \times 100\% = 1\%$. From this example, we can conclude that percentage rank is not a practical evaluation metric for the software systems of today, possibly consisting of millions lines of code, which is also confirmed by the authors' preliminary study. To observe whether, and to what extent, the community has improved in this area, we include all papers that adopt **absolute evaluation metrics**.

Observation 2 and Implication 2 mention that future research should focus on searching through suspicious statements and providing more contextual information such that it is easier for the user to interpret the fault localization results. This implication has also been confirmed by Minelli *et al.* [25], who performed an empirical study that strongly suggests that the importance of program comprehension has been significantly underestimated by prior research. In our opinion, *searching* through the fault diagnosis is too specific, and ultimately focuses on improving result comprehension.

Therefore, to generalize this implication, we include studies in our survey that focus on **result comprehension**.

The third implication suggests future research to focus on creating complete ecosystems. Therefore, in this survey, we include work that propose or improve existing **ecosystems**.

Finally, Parnin and Orso mention that more research has to be performed in the form of user studies, as they did themselves. Hence, we give an overview of **user studies** in the field of fault localization techniques.

IV. Impact of Parnin and Orso's Study

In this section, we discuss the selection methodology and give an overview of studies for each research direction proposed by Parnin and Orso as discussed in Section III-B. In Table I, an overview of studies is provided sorted by the year of publication, indicating the problems that each study tackles.

A. Selection

In this survey, the initial body of literature comprises work that refer to Parnin and Orso's study, amounting to 104 published studies on Scopus[1] at the time of writing. These papers were obtained with Scopus because it only consists of peer-reviewed papers. Next, papers that are not written in English or accessible are removed from the set of literature. Finally, we read the abstract and relevant sections that refer to Parnin and Orso of each study, and we determined if it attempts to solve one of the observations or implications made by Parnin and Orso. This results in a body of literature of 19 papers. Studies, that mention Parnin and Orso's work but do not consider their findings, referred to Parnin and Orso's study for various reasons: (1) Parnin and Orso's findings are mentioned as a potential threat to validity, (2) the authors are referred to as related work.

B. Absolute Evaluation Metrics

Jin and Orso [28] proposed F^3 that extends BugRedux, a technique for reproducing failures observed in the field, with fault localization capabilities. In their study, the authors evaluate F^3 using **wasted effort**, indicating the number of non-faulty components that have to be inspected on average before a faulty component is found in the diagnostic report. In their study, the authors use the following formula to compute wasted effort.

$$\textit{wasted effort} = m + n + 1$$

where m is the number of non-faulty components that are assigned a strictly higher suspiciousness score than the faulty component, and n is the number of non-faulty components that are assigned an equal suspiciousness score as the faulty component. Note that the formula used to compute wasted effort can vary. For example, Laghari *et al.* [37] compute wasted effort as follows.

$$\textit{wasted effort} = m + (n+1)/2$$

The wasted effort is also used in [26], [31], [39], [40].

[1]https://www.scopus.com/

Lo *et al.* [32] proposed an approach to combine multiple spectrum-based fault localization techniques, namely Fusion Localizer. In their study, the authors investigate multiple approaches to score normalization, technique selection, and data fusion, resulting in twenty variants of Fusion Localizer. In the evaluation of the proposed Fusion Localizer variants, the authors make use of **accuracy at n** (acc@n), which indicates the number of bugs that can be diagnosed when inspecting the top n components in the ranked list. This metric is also used in [33].

Le *et al.* [38] proposed a new automated debugging technique, called Savant, that employs learning-to-rank, using changes in method invariants and suspicious scores, to diagnose faults. To evaluate Savant, the authors make use of three absolute rank-based metrics, namely acc@n, wef@n, and MAP. **Wasted effort at n** (wef@n) is a variation of wasted effort that computes the wasted effort within the top n components of the ranked list. The **Mean Average Precision** (MAP) [46] metric is widely used in information retrieval. MAP is computed by computing the mean of the average precisions (APs), that is computed as follows:

$$AP = \frac{1}{M}\sum_{i=1}^{N} P(i)rel(i)$$

where M is the number of total faulty program components, N is the total number of components in the ranked list, $P(i)$ is the precision at the i^{th} component in the diagnosis report, and $rel(i)$ is a binary indicator indicating whether component i is faulty, i.e. relevant. The precision at position k ($P(k)$) is computed as follows:

$$P(k) = \frac{\textit{number of faulty components within top k}}{k}$$

Finally, MAP is computed by averaging the average precisions of each produced ranked list.

Laghari *et al.* [37] also make use of wef@n to evaluate the performance of their proposed technique: patterned spectrum analysis. In their study, they use method call patterns, which are obtained by adopting the closed itemset mining algorithm [47], as hit-spectrum to perform SBFL.

Wen *et al.* [43] proposed an IRBFL technique, called Locus. Locus is able to locate bugs at both the software change and source file level — the latter is a common granularity used in IRBFL techniques. It leverages the information of bug reports, source code changes, and change history to localize suspicious hunks. In the evaluation of Locus, the authors made use of three metrics: Top@n, MRR, and MAP. Top@n reports how many bugs are diagnosed in the top n suspicious code entities, and is therefore identical to acc@n. The **Mean Reciprocal Rank** (MRR) [48] is another metric used in information retrieval to evaluate the performance. The formula of MRR is as follows:

$$MRR = \frac{1}{Q}\sum_{i=1}^{Q}\frac{1}{rank_i}$$

where Q is the number of queries, i.e. the number of performed fault diagnoses, $rank_i$ is the position of the first true positive

TABLE I: Overview of studies surveyed in this work.

Year	Author	Title	Evaluation metric	Result comprehension	Ecosystem	User study
2013	Campos *et al.* [26]	Entropy-based test generation for ...	•			
2013	Gouveia *et al.* [27]	Using HTML5 visualizations in ...		•	•	•
2013	Jin and Orso [28]	F3: fault localization for field failures	•			
2013	Pastore and Mariani [29]	AVA: supporting debugging with ...		•	•	
2013	Qi *et al.* [30]	Using automated program repair ...	•			
2014	Liu *et al.* [31]	Simulink fault localization: an ...	•			
2014	Lo *et al.* [32]	Fusion fault localizers	•			
2014	Wu *et al.* [33]	CrashLocator: locating crashing ...	•			
2014	Zuddas *et al.* [34]	MIMIC: Locating and ...		•		
2015	Wang *et al.* [35]	Evaluating the usefulness of ...				•
2016	Kochhar *et al.* [36]	Practitioners' expectations on ...				•
2016	Laghari *et al.* [37]	Fine-tuning spectrum based fault ...	•			
2016	Le *et al.* [38]	A learning-to-rank based fault ...	•			
2016	Li *et al.* [39]	Iterative user-driven fault localization	•	•		
2016	Li *et al.* [40]	Towards more accurate fault ...	•			
2016	Wang and Huang [41]	Weighted control flow subgraph to ...		•		
2016	Wang and Liu [42]	Fault localization using disparities ...		•		
2016	Wen *et al.* [43]	Locus: locating bugs from software ...	•	•		
2016	Xia *et al.* [44]	"Automated Debugging Considered ...				•
2016	Xie *et al.* [45]	Revisit of automatic debugging via ...				•

diagnosed component. This metric evaluates the ability to locate the first faulty component.

Qi *et al.* [30] analyzed the effectiveness of automated debugging techniques from the perspective of fully automated program repair. The automated program repair process can be divided into three phases: fault localization, patch generation, and patch validation. With this in mind, the authors proposed the NCP metric, the **number of candidate patches** that are generated in the patch generation phase. Intuitively, a well-performing fault localization technique would require a lower number of generated candidate patches because the faulty component is ranked higher in the diagnosis report.

To summarize, we observe that studies in software fault localization have adopted absolute evaluation metrics since Parnin and Orso's study, namely wasted effort, accuracy at n, wasted effort at n, mean average precision, mean reciprocal rank, and the number of candidate patches. Moreover, wasted effort is slowly becoming the standard to evaluate the fault localization performance.

C. Result Comprehension

Gouveia *et al.* [27] implemented GZoltar, a plug-and-play plugin for the Eclipse Integrated Development Environment (IDE) that performs fault localization and visualizes the suspiciousness of program components. Specifically, GZoltar visualizes the results in three different ways: sunburst, vertical partition, and bubble hierarchy. The authors found evidence that the visualizations aid the developer in finding the root cause of a bug, which we discuss in more depth in Section IV-E.

Wang and Liu [42] presented an automated debugging technique using disparities of dynamic invariants, named FDDI. FDDI uses a spectrum-based fault localization technique to localize the most suspicious functions. Then, FDDI uses Daikon to infer dynamic invariant sets for the passing and failing test suites. Finally, FDDI performs a disparity analysis between the two invariant sets and generates a debugging report that comprises suspect statements and variables. The variables are extracted from the disparity, which could assist users in finding and understanding the root cause of a bug.

As mentioned in Section IV-B, Wen *et al.* [43] performed fault localization based on software changes, resulting in a list of suspicious change hunks. The advantage of outputting change hunks is twofold. First, the time spent on bug triaging is reduced because developers are linked to change hunks. The authors showed in an empirical study that 70% to 80% of the bugs are fixed by the developer who introduced the bug. A possible explanation for this is that the developer, who introduced the bug, is familiar with the code. Second, change hunks consist of contextual lines (unchanged lines), changed lines, and a corresponding commit description, providing the developer with contextual information to understand the diagnosed change hunks.

Wang and Huang [41] proposed the use of weighted control flow subgraphs (WCFSs) to provide contextual information on the suspicious components in the diagnosis report. The WCFSs are constructed from the execution traces collected during the execution of the test suite, which are also used to construct the activity matrix for SBFL. For each suspicious component in the diagnosis report, the authors allow the developer to display the associated WCFS. This enables the developer to navigate or search the diagnosed components in a more natural manner.

Li *et al.* [39] proposed an SFL technique, named Swift, that involves the developer in the fault localization process.

Swift performs SBFL but instead of displaying a ranked list, it guides the developer through the diagnosis report by showing the developer a query for the most suspicious method. The query consists of the input and output of the method invocation, which the developer has to validate by marking it as correct or incorrect. Then, the fault probabilities are modified accordingly and Swift generates a new query for the next most suspicious method.

Zuddas *et al.* [34] proposed a prototype tool, called MIMIC, that identifies potential causes of a failure. MIMIC is able to do this by performing four steps: execution synthesis, monitoring points detection, anomaly detection, and filtering. The output of MIMIC does not simply consist of suspicious statements but, instead, provides code locations, their supposedly correct behavior model, and the actual values that violate the generated behavioral model. In an empirical study, the authors show that MIMIC can effectively detect failure causes.

Pastore and Mariani [29] proposed AVA, a fault localization technique that generates an explanation about why diagnosed components are considered suspicious. It does this by comparing execution traces to a finite state automaton (FSA), which is commonly inferred from successful program executions. The suspicious components are detected using KLFA [49]. KLFA is also able to classify the difference between the actual and expected behavior according to a set of defined patterns, e.g. delete, insert, replace, etc. The classification and the suspicious components are then displayed to the developer such that the developer is able to determine whether a suggested component is truly faulty.

To summarize, several studies have focused on improving result comprehension in software fault localization. However, most studies evaluate their approach with a case study, rather than with a study involving actual users.

D. Ecosystems

As mentioned in Section IV-C, Gouveia *et al.* [27] have developed the GZoltar toolset, which is available as an Eclipse plug-in. The toolset localizes faults by employing a spectrum-based fault localization technique, namely Ochiai [50], that takes as input the coverage information of executed test cases. By performing SBFL the toolset produces a ranked list of suspicious program components. In addition, as a response to the findings of Parnin and Orso [1], the authors have improved the plug-in by extending the toolset with visualization capabilities.

Another tool that was created as a response to Parnin and Orso's findings is AVA. AVA [29] consists of two main components: the AVA-core library and the AVA-Eclipse Eclipse plug-in. The AVA-core library implements an API that can be invoked from third-party programs to generate interpretations from anomalies. The Eclipse plug-in provides the developer with a GUI in Eclipse to perform debugging using AVA.

We observe that the SFL research community has not yet put a lot of effort in creating tools that can be used by developers. Therefore, we suggest that more effort should be spent on developing tools that facilitate automated debugging techniques.

E. User Studies

To verify the effectiveness of the visualizations generated by GZoltar in practice, Gouveia *et al.* [27] performed a user study. The experiment involved 40 participants divided into two groups: a control group that is only allowed to make use of the default debugging tools provided by the Eclipse IDE and a test group that has to use GZoltar for debugging. The user experiment showed evidence that the mean time of completing the debugging task of the test group is significantly shorter than the mean time of the control group. In fact, the test group took on average 9 minutes and 17 seconds less than the control group to find the injected fault.

Xie *et al.* [45] reproduced a similar user study to Parnin and Orso's work [1] that differs in the size of involved participants and debugging tasks, namely 207 participants and 17 debugging tasks. The experiments are performed on a platform, called Mooctest, that is able to localize faults, track user behavior such as mouse position, and analyze produced logs. The main finding of the user study is that, regardless of the accuracy, spectrum-based fault localization does not reduce time spent in debugging a fault. Also, inaccurate fault localization results may even lengthen the debugging process. Based on these results, the authors corroborated the findings of Parnin and Orso — more research should be performed on result comprehension.

Kochhar *et al.* [36] performed a user study by means of a survey involving 386 practitioners from more than 30 countries. In the survey, the authors found that practitioners have high thresholds for adopting automated debugging techniques. A comparison between the expectations of practitioners and the state-of-the-art fault localization techniques showed that research should primarily focus on improving reliability, scalability, result comprehension, and IDE integration, such that practitioners' expectations can be met.

Wang *et al.* [35] evaluated IR-based fault localization techniques by means of an analytical study and one involving human subjects. In the analytical study, Wang *et al.* showed evidence that the performance of IRBFL techniques is determined by the quality of bug reports. However, the authors also found that a large portion of the bug reports does not contain enough identifiable information, and therefore IRBFL techniques are less effective in the majority of cases. In the user experiment, the authors found evidence that IRBFL techniques are helpful when bug reports do not contain rich information but are unlikely to be effective otherwise.

Xia *et al.* [44] reproduced a similar user study to the work of Parnin and Orso as well as Xie *et al.*. Their user study involved 36 professionals and 16 real bugs from 4 reasonably large open source projects. However, unlike Parnin and Orso and Xie *et al.*, Xia *et al.* show evidence that SBFL does reduce time spent debugging.

To summarize, we observe that the research community has performed a couple of user studies to understand the users' needs. However, besides the mentioned user studies, almost no study evaluates their technique with a user study, which

would be particularly useful in determining its effectiveness.

V. Research Implications

In Section IV, we observed that more studies are adopting an absolute metric to measure the performance of SFL techniques. In particular, we see that wasted effort is slowly becoming the standard for SFL evaluation.

However, Parnin and Orso have shown that outputting an accurate diagnostic report does not necessarily result in less time spent on debugging since *perfect bug understanding* is an assumption that does not hold in practice. Hence, we conclude that *more studies should focus on improving result comprehension, i.e. in assisting the developer in making sense of the SFL output.*

Although there are a few studies that propose a solution for better result comprehension, almost none of them evaluate their solution with a user study. In case of GZoltar, its visualizations are evaluated with a user study and the authors have shown evidence that debugging with GZoltar reduces time spent on debugging. However, while the debugging time is reduced, no study has yet analyzed the debugging process with an SFL tool in depth. For example, does a developer run an SFL tool multiple times before fixing a bug? Or is a bug fixed after the first analysis? How many suspicious locations identified by an SFL tool are typically visited by the developer? For what reasons? To answer such questions, *we need a theory describing successful use of software fault localization techniques* — developing such a theory calls for extensive qualitative studies [51] with developers interacting with such techniques.

To perform studies that focus on how to improve result comprehension, we need tooling. Parnin and Orso have pointed out that the SFL community needs to focus on tooling, but in our survey we have not seen significant advancements in this area. Although creating a tool requires a lot of effort, we are not able to push forward SFL research if we do not spend time on developing SFL tools. Therefore, *we call for an open source community for SFL tooling such that development efforts can be distributed among researchers.* Creating an open source community for SFL also has the benefit that replication studies are easier to perform and therefore allows comparisons to be made. This tooling environment should also provide an integrated, rich source of additional data that diagnostic techniques can leverage. Using historical data to assess multiple-fault prevalence [52] and constructing prediction models from issue trackers to improve SFL diagnoses [53] are two examples of work benefiting from such integration.

When tooling exists we are able to perform more user studies. Since Parnin and Orso's study, only five user studies [27], [35], [37], [44], [45] have been performed. A possible cause is that tooling does not yet exist and requires a lot of effort to develop. However, *user studies are essential to fully understand how to improve the current state of SFL techniques, and how to make SFL techniques being adopted in the software development cycle.*

VI. Conclusion

In the past two decades, substantial effort has been put in improving software fault localization techniques. However, Parnin and Orso were one of the first to perform a user study and found that the assumptions made by SFL techniques do not actually hold in practice. As an example, the common assumption of *perfect bug understanding* does not hold in practice. For this reason, Parnin and Orso suggested a number of research directions which we generalized into absolute evaluation metrics, result comprehension, ecosystems, and user studies.

In our survey, we found that Since Parnin and Orso's study, the SFL research community is slowly adopting the *absolute* evaluation metric. Furthermore, it has proposed several techniques to improve *result* comprehension. Unfortunately, substantially less effort has been put in developing ecosystems and performing user studies, which play essential roles in closing the gap between research and practice.

Based on these observations, we recommend the SFL research community to focus on creating an ecosystem that can be used by developers during debugging activities. Such an ecosystem can serve as a framework for SFL such that researchers can easily implement their techniques in the framework and evaluate them in user studies. While current studies mostly evaluate their SFL technique using absolute metrics, actual adoption requires insights that can only be obtained from user studies of automated debugging techniques used in practice.

Acknowledgments

This material is based upon work supported by the scholarship number SFRH/BD/95339/2013 and project POCI-01-0145-FEDER-016718 from Fundação para a Ciência e Tecnologia (FCT), by ERDF COMPETE 2020 Programme, by EU Project STAMP ICT-16-10 No.731529 and by 4TU project "Big Software on The Run".

References

[1] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 199–209.

[2] J. Robbins, *Debugging Applications for Microsoft .NET and Microsoft Windows*. Microsoft Press, 2003.

[3] B. Beizer, *Software testing techniques*. Dreamtech Press, 2003.

[4] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, "Reversible debugging software," *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, 2013.

[5] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, Aug 2016.

[6] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," in *Software EngineeringESEC/FSE'97*. Springer, 1997, pp. 432–449.

[7] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization techniques," University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-16-08-03, Sep. 2016.

[8] M. D. Weiser, "Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method," Ph.D. dissertation, University of Michigan, Ann Arbor, MI, USA, 1979, aAI8007856.

[9] B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, 1988.

[10] J. S. Collofello and L. Cousins, "Towards automatic software fault location through decision-to-decision path analysis," *Managing Requirements Knowledge, International Workshop on*, vol. 00, p. 539, 1987.

[11] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization for fault localization," in *Proceedings of ICSE 2001 Workshop on Software Visualization, Toronto, Ontario, Canada*. Citeseer, 2001, pp. 71–75.

[12] R. Davis, "Diagnostic reasoning based on structure and behavior," *Artificial intelligence*, vol. 24, no. 1, pp. 347–410, 1984.

[13] R. Reiter, "A theory of diagnosis from first principles," *Artificial intelligence*, vol. 32, no. 1, pp. 57–95, 1987.

[14] J. De Kleer and B. C. Williams, "Diagnosing multiple faults," *Artificial intelligence*, vol. 32, no. 1, pp. 97–130, 1987.

[15] C. Mateis, M. Stumptner, and F. Wotawa, "Debugging of java programs using a model-based approach," in *Proceedings of the Tenth International Workshop on Principles of Diagnosis*, 1999.

[16] C. Mateis, M. Stumptner, D. Wieland, and F. Wotawa, "Jade-ai support for debugging java programs." in *ictai*, 2000, p. 62.

[17] C. D. Manning, P. Raghavan, H. Schütze *et al.*, *Introduction to information retrieval*. Cambridge university press Cambridge, 2008, vol. 1, no. 1.

[18] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 2011, pp. 43–52.

[19] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 14–24.

[20] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, 2010.

[21] ——, "Source code retrieval for bug localization using latent dirichlet allocation," in *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on*. IEEE, 2008, pp. 155–164.

[22] D. Poshyvanyk, A. Marcus, V. Rajlich, Y.-G. Gueheneuc, and G. Antoniol, "Combining probabilistic ranking and latent semantic indexing for feature identification," in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*. IEEE, 2006, pp. 137–148.

[23] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*. IEEE, 2004, pp. 214–223.

[24] H. A. Souza, M. L. Chaim, and F. Kon, "Spectrum-based software fault localization: A survey of techniques, advances, and challenges," *arXiv preprint arXiv:1607.04347*, 2016.

[25] R. Minelli, A. Mocci, M. Lanza, and T. Kobayashi, "Quantifying program comprehension with interaction data," in *Quality Software (QSIC), 2014 14th International Conference on*. IEEE, 2014, pp. 276–285.

[26] J. Campos, R. Abreu, G. Fraser, and M. d'Amorim, "Entropy-based test generation for improved fault localization," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 257–267.

[27] C. Gouveia, J. Campos, and R. Abreu, "Using html5 visualizations in software fault localization," in *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*. IEEE, 2013, pp. 1–10.

[28] W. Jin and A. Orso, "F3: fault localization for field failures," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 2013, pp. 213–223.

[29] F. Pastore and L. Mariani, "Ava: Supporting debugging with failure interpretations," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, March 2013, pp. 416–421.

[30] Y. Qi, X. Mao, Y. Lei, and C. Wang, "Using automated program repair for evaluating the effectiveness of fault localization techniques," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 2013, pp. 191–201.

[31] B. Liu, S. Nejati, L. C. Briand, and T. Bruckmann, "Simulink fault localization: an iterative statistical debugging approach," *Software Testing, Verification and Reliability*, 2016.

[32] D. Lo, X. Xia *et al.*, "Fusion fault localizers," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 127–138.

[33] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, "Crashlocator: Locating crashing faults based on crash stacks," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 204–214. [Online]. Available: http://doi.acm.org/10.1145/2610384.2610386

[34] D. Zuddas, W. Jin, F. Pastore, L. Mariani, and A. Orso, "Mimic: Locating and understanding bugs by analyzing mimicked executions," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 815–826. [Online]. Available: http://doi.acm.org/10.1145/2642937.2643014

[35] Q. Wang, C. Parnin, and A. Orso, "Evaluating the usefulness of ir-based fault localization techniques," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771797

[36] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 165–176. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931051

[37] G. Laghari, A. Murgia, and S. Demeyer, "Fine-tuning spectrum based fault localisation with frequent method item sets," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 274–285.

[38] T.-D. B Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 177–188.

[39] X. Li, M. d'Amorim, and A. Orso, "Iterative user-driven fault localization," in *Haifa Verification Conference*. Springer, 2016, pp. 82–98.

[40] A. Li, Y. Lei, and X. Mao, "Towards more accurate fault localization: An approach based on feature selection using branching execution probability," in *Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on*. IEEE, 2016, pp. 431–438.

[41] Y. Wang and Z. Huang, "Weighted control flow subgraph to support debugging activities," in *Software Quality, Reliability and Security Companion (QRS-C), 2016 IEEE International Conference on*. IEEE, 2016, pp. 131–134.

[42] X. Wang and Y. Liu, "Fault localization using disparities of dynamic invariants," *Journal of Systems and Software*, vol. 122, pp. 144–154, 2016.

[43] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 262–273. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970359

[44] X. Xia, L. Bao, D. Lo, and S. Li, "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," in *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE, 2016, pp. 267–278.

[45] X. Xie, Z. Liu, S. Song, Z. Chen, J. Xuan, and B. Xu, "Revisit of automatic debugging via human focus-tracking analysis," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 808–819.

[46] C. D. Manning, H. Schütze *et al.*, *Foundations of statistical natural language processing*. MIT Press, 1999, vol. 999.

[47] M. J. Zaki and C.-J. Hsiao, "Charm: An efficient algorithm for closed itemset mining." in *SDM*, vol. 2. SIAM, 2002, pp. 457–473.

[48] E. M. Voorhees *et al.*, "The trec-8 question answering track report." in *Trec*, vol. 99, 1999, pp. 77–82.

[49] L. Mariani and F. Pastore, "Automated identification of failure causes in system logs," in *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*, Nov 2008, pp. 117–126.

[50] R. Abreu, P. Zoeteweij *et al.*, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 2006, pp. 39–46.

[51] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Trans. Software Eng.*, vol. 25, no. 4, pp. 557–572, 1999. [Online]. Available: https://doi.org/10.1109/32.799955

[52] A. Perez, R. Abreu, and M. d'Amorim, "Prevalence of single-fault fixes and its impact on fault localization," in *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017*, 2017, pp. 12–22.

[53] A. Elmishali, R. Stern, and M. Kalech, "Data-augmented software diagnosis," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, 2016, pp. 4003–4009.