2016 IEEE/ACM 38th IEEE International Conference on Software Engineering

Revisit of Automatic Debugging via Human Focus-tracking Analysis

Xiaoyuan Xie
State Key Lab of Software Engineering, Computer School
Wuhan University
xxie@whu.edu.cn

Zicong Liu
State Key Lab of Novel Software Technology
Nanjing University
zicong.liu.nju@gmail.com

Shuo Song
State Key Lab of Novel Software Technology
Nanjing University
songshuosz@gmail.com

Zhenyu Chen*
State Key Lab of Novel Software Technology
Nanjing University
zychen@nju.edu.cn

Jifeng Xuan
State Key Lab of Software Engineering, Computer School
Wuhan University
jxuan@whu.edu.cn

Baowen Xu
State Key Lab of Novel Software Technology
Nanjing University
bwxu@nju.edu.cn

ABSTRACT

In many fields of software engineering, studies on human behavior have attracted a lot of attention; however, few such studies exist in automated debugging. Parnin and Orso conducted a pioneering study comparing the performance of programmers in debugging with and without a ranking-based fault localization technique, namely Spectrum-Based Fault Localization (SBFL). In this paper, we revisit the actual helpfulness of SBFL, by addressing some major problems that were not resolved in Parnin and Orso's study. Our investigation involved 207 participants and 17 debugging tasks. A user-friendly SBFL tool was adopted. It was found that SBFL tended not to be helpful in improving the efficiency of debugging. By tracking and analyzing programmers' focus of attention, we characterized their source code navigation patterns and provided in-depth explanations to the observations. Results indicated that (1) a short "first scan" on the source code tended to result in inefficient debugging; and (2) inspections on the pinpointed statements during the "follow-up browsing" were normally just quick skimming. Moreover, we found that the SBFL assistance may even slightly weaken programmers' abilities in fault detection. Our observations imply interference between the mechanism of automated fault localization and the actual assistance needed by programmers in debugging. To resolve this interference, we provide several insights and suggestions.

CCS Concepts

•Software and its engineering → Software testing and debugging;

*Corresponding author.

ICSE '16, May 14-22, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2884781.2884834

Keywords

Automated debugging, spectrum-based fault localization, user studies, attention tracking, navigation pattern, fault comprehension

1. INTRODUCTION

It is widely known that software engineering is a human-centric discipline [29]. Studies on human behavior during the software life cycle have attracted extensive attention in the community [6, 14, 21, 30]. Such studies usually aimed to reveal how people actually behave and perform in various software engineering activities, in order better facilitate these activities.

Software fault detection is one of the most human-intelligence-intensive tasks in which analysis of human behavior is essential. Uwano et al. [33] took the first step, performing eye-tracking analysis in static code review, where a particular pattern, namely "scan" was identified. "Scan" characterizes how a reviewer navigates the entire source code before carefully inspecting each code line. On the other hand, Bednarik [2] shifted the focus from static code review to manual debugging[1], specifically, how programmers dealt with various information sources was investigated.

Apart from static code review and manual debugging, there is another large family of fault detection techniques, namely automated debugging, which has been widely investigated over the past two decades. In this field, researchers have been mainly focused on proposing new techniques and investigating their performance [1, 12, 20, 36–38] without paying much attention to human factors. However, the results provided by current automated debugging techniques are only approximate, and their correctness is generally not guaranteed. As a consequence, these results still require being understood and verified by the person who performs the repair. Therefore, human behavior is critical and worth investigating in these techniques.

Parnin and Orso realized this problem and conducted a study which compared the actual performance of developers in debugging with and without a lightweight ranking-based fault localization technique [21], where they explicitly indicated the importance of human factors in determining the helpfulness of this technique. As a preliminary study, this work reveals several potential aspects

[1]We refer to "debugging" in this paper as the dynamic fault elimination process using test execution information, in contrast to the static code review.

that should be further investigated.

First, a more comprehensive experiment is required to obtain a statistically conclusive result. Secondly, the helpfulness of a better automated debugging tool is worth studying. Thirdly, the impact from the accuracy of fault localization needs to be re-examined by considering the suggested measure of "absolute ranking" [21]. Fourthly, although it has been generally accepted that there is no "perfect bug[2] detection", we are interested to know whether automated debugging has any impact on human ability in comprehending the source code and identifying the bugs. Finally, we still do not know what particular patterns programmers may follow in navigating the source code during automated debugging and how these patterns affect debugging efficiency.

Therefore, in this study, we revisit the helpfulness of automated debugging by addressing the above points. We are most interested in identifying the reasons behind the observed results via a quantitative focus-tracking analysis, which refers to tracking and analyzing the focus of attention of programmers. In this paper, we answer three Research Questions (RQ) from an experiment with 207 participants and 17 debugging tasks which were categorized according to their fault localization accuracies. Our experiment used the same ranking-based fault localization technique that Parnin and Orso used in [21], i.e. Spectrum-Based Fault Localization (SBFL). Each participant was required to debug two programs with and without SBFL, respectively. A platform, namely Mooctest [19], with user-friendly SBFL assistance was implemented, which can colorize the suspicious code lines and record all of the operations performed by the participants during debugging, such as file opening, cursor movement and file changing. These recorded operations were used to analyze the focus of attention of the participants and to reveal their code navigation patterns.

We found that an accurate localization result was helpful in completing the debugging tasks. However, surprisingly, we also observed that SBFL was not helpful in improving the efficiency of debugging and sometimes even made it worse. By analyzing the operation logs recorded by Mooctest, we found some navigation patterns. In debugging with SBFL, programmers typically started with a "first scan", i.e. a skim of the entire or part of the code, before inspecting the most suspicious code highlighted by the platform, and then switched between the most suspicious code and others. A quantitative analysis showed that (1) a short "first scan" tended to result in inefficient debugging; and (2) inspections on the pinpointed statements during the "follow-up browsing" on the source code after the "first scan" were normally just quick skimming. Moreover, we found that the SBFL assistance in Mooctest, especially for cases with inaccurate results, may even slightly weaken programmers' abilities in comprehending the code and identifying the faults.

Our analysis confirms the importance of code comprehension. Also it implies that presenting code together with fault ranking information, which intends to help programmers focus on the suspicious code as quickly as possible without wasting time on other parts of the program, may actually interfere with the real assistance needed by programmers in debugging. This interference is most likely the reason for the decrease in efficiency. We conjecture that the interference is due to an inappropriate application of the technique. We further give some insights and suggestions on how to properly perform automated debugging, in order to really benefit from it.

The contributions of this paper are summarized as follows.

- We conducted a comparison of programmers' performance between debugging with and without SBFL, where 207 participants and 17 debugging tasks were involved and a user-friendly debugging platform was adopted. We also analyzed the impact from different levels of SBFL accuracy by considering "absolute ranking".
- We performed analysis to track the focus of attention of our participants, where two important code navigation patterns, namely, "first scan" and "follow-up browsing", were observed. And their relations with the efficiency of SBFL-assisted debugging were revealed.
- We reconfirmed the importance of code comprehension with supportive evidence. Our analysis also revealed that automated fault localization may interfere with programmers' code comprehension and fault identification. To help programmers really benefit from automated debugging, we provided some insights and suggestions.

[2]In this paper, we will use "bug" and "fault" interchangeably.

2. RELATED WORK

2.1 Analyses of human behavior in software engineering

Human behavior has been considered to have a major impact on the entire software life cycle [9]. Understanding how people behave and perform in various software engineering activities can help to answer why a particular technique is or is not effective.

Crosby and Stelovsky [6] first investigated the influence of programming experience on viewing a short but complex algorithm via an eye-tracking technique. Sharif and his colleagues conducted eye-tracking studies on UML diagram comprehension [27]. They also performed similar investigation about the impact of identifier style on code comprehension [3]. Rodeghero et al. [23] tracked developers' focus of attention to provide evidence on how humans chose the keywords of a program for source code summarization. And Lewis et al. [18] evaluated how well a notable bug predication algorithm, FixCache, helped developers.

In static code review, Uwano et al. [33] took the first step in adopting eye-tracking techniques to explain the performance of various reviewing strategies. Uwano et al. identified a "scan" pattern in static code review, i.e., people usually have a preliminary reading of the entire code at the beginning of their review. They found that the longer a reviewer scanned the code, the more efficiently he could find the fault in code review. This finding was later confirmed by Sharif et al. [26] via more comprehensive empirical studies.

Concerning debugging, Stein and Brennan [30] conducted an interesting analysis, which demonstrated that novice programmers can debug more efficiently after viewing the eye gaze traces from other professional programmers. Ko et al. [13] tracked developers actions such as "searching", "navigating", and "editing" to answer: "how do developers seek, relate, and collect relevant information" in debugging tasks. There were also studies that aimed at predicating programmers' code navigation behavior [16, 22]. However, all of these works were about manual debugging; for automated debugging, to the best of our knowledge, there are still very few similar studies.

2.2 Analyses on automated debugging with human factors

It is widely accepted that debugging is extremely time-consuming. In order to speed-up this process, many automated debugging techniques have been proposed over the past two decades.

One type of techniques utilizes human intelligence to facilitate debugging, for example one is called Whyline implemented by Ko et al. [14]. Ko et al. invented a new debugging paradigm, namely,

"Interrogative Debugging", which allowed programmers "communicate" with the software, by asking "why did" questions. Answers given by Whyline can guide programmers to find the cause of the failure and fix the bug.

Another family of automated debugging techniques aims to reduce human involvement by providing possible fault locations. One classic idea is to use program dynamic slicing, which can effectively isolate statements involved in the calculation of the observed failure. This method was first proposed by Korel and Laski [15], followed by a series improved versions [28, 40]. Another idea is to model a fault localization problem into an information retrieval task, where bug reports are treated as queries and potential faults are the most matchable code areas [24]. Besides, there is a widely studied method, namely Spectrum-Based Fault Localization (SBFL), which utilizes various program spectra and testing results to evaluate the suspiciousness for each program entity. All entities are then ranked in descending order and are provided to programmers as the suggested fault locations [1, 12, 37].

However, the fault locations suggested by this family of techniques still require programmers' comprehension and verification to repair the software. In other words, although they reduce manual efforts, these techniques cannot completely eliminate human involvement. Hence, whether they are really helpful to programmers is well worth investigating. However, most of these studies focused on how to improve the accuracy of fault localization; and very few have taken human factors into account.

Weiser and Lyle [35] did the first comparison of developers' performance between debugging with and without program slicing, where no significant improvement was found by using slicing. Francel and Spencer [7] also investigated the helpfulness of program slicing to programmers in their code comprehension and debugging abilities. In contrast, their experiments indicated that slicing improved programmers' performance. Tao et al. [31] studied the effectiveness of human debugging via machine-generated patches. A recent study by Wang et al. investigated how well information retrieval techniques help developers to locate bugs [34].

In the field of SBFL that has been regarded as a light-weight and practical fault localization technique, even fewer related works have been reported. Parnin and Orso [21] did a pioneering study to answer the question "can automated debugging help programmers", where the actual performance of participants in debugging with and without SBFL was compared. They showed that SBFL can be helpful in some cases, especially for experienced programmers. They also reported some surprising observations, such as "changes in the ranking do not produce any significant impact on programmers' performance", "programmers may not follow the provided ranking list", and pointed out that the "perfect bug detection" is generally unrealistic. Besides, some suggestions were given based on their observations to better improve the SBFL tools.

3. MOTIVATION AND RESEARCH QUESTIONS

The study by Parnin and Orso [21] provides a new perspective from which to re-examine SBFL. Many aspects were revealed to be in need of further investigation such as:

(1) Large-scale experiments in terms of the number of participants and debugging tasks should be conducted. The experiment in [21] only involved 34 programmers and two faulty programs. Therefore, it is necessary to have larger-scale experiments for a more statistically conclusive result.

(2) As suggested in [21], programmers' performance with a user-friendly tool is worth studying. Parnin and Orso [21] provided a plug-in for SBFL which presented the localization results simply in a ranking list. For example, two "for-loop" predicates are ranked first and second in the list. Then, a result presented to a programmer looks like:

$$1 : \text{"for (int } i = 0; i < N; i\text{++)"}$$
$$2 : \text{"for (int } i = 0; i < M; i\text{++)"}$$
$$3 : \ldots (\text{other statements})$$

Clearly, such reordered statements are incomprehensible. Parnin and Orso [21] alleviated this problem by providing navigation from each statement in the list to its original location. They have done this on purpose, because their goal was to investigate the impact purely from the ranking list. However, it is also interesting to know whether a more easy-to-follow result can lead to similar or better performance. Actually, Gouveia et al. have considered programmers' performance with their proposed dynamic graphical localization reports [10]. However, since the human factor was not the focus of that study, experiments were still very preliminary.

In addition, facilities of test execution can be improved. In [21], for each program, only one failed test case was provided in a descriptive way, rather than in an executable format[3]. More importantly, the failure information in one of their subject programs clearly indicated the lines throwing the exceptions, upon which programmers were more likely to rely.

(3) Impact on debugging performance from SBFL accuracy with respect to the "absolute ranking", needs to be re-examined. As conjectured by Parnin and Orso [21], programmers may not follow the ranking after inspecting a few irrelevant statements, which could be one possible explanation for their "no impact" observation, since neither of their changes on the list has ever had the fault ranked top.

(4) Whether SBFL could affect fault detection, i.e. comprehend and identify the fault, is still unknown. It has been generally accepted that there is no "perfect bug detection". Simply inspecting a fault is not sufficient for immediately detecting it. However, using the SBFL assistance and detecting the fault may not be completely independent of one another. Therefore, it is interesting to know whether the SBFL assistance has any impact on fault detection.

(5) Last but most importantly, an in-depth analysis that tracks programmers' focus of attention to reveal their code navigation pattern is needed. Parnin and Orso provided some preliminary discussion on how developers navigate the program when debugging. However it will be more convincing and informative if some visualized and quantitative analysis can be provided.

In summary, there are still many questions to be answered, which can help to reveal why SBFL is or is not helpful, and how to improve it. Therefore, in our study, we revisit the performance of SBFL by considering human factors to address the above points. We recruited 207 participants to perform debugging on 17 faulty programs. A platform, namely Mooctest, with better SBFL assistance than the plug-in in [21] was implemented. It adopted Jones et al.'s [12] idea to colorize the code according to their suspiciousness without changing their positions, such that the risk can be pinpointed but the integrity and readability of the program can still be preserved. Each debugging task was associated with a set of JUnit test cases that were executable, and the execution results of these test cases were linked to the SBFL colorization. Our investigation was no longer solely on the helpfulness of the suggested

[3]The plug-in hard-codes the SBFL ranking in a static XML file, which is not related to the given failed test case.

ranking list, but of the entire debugging tool. To learn the impact from SBFL accuracy, we compared debugging with "good" (if the faulty statement is ranked top) and "bad" (otherwise) SBFL results. More importantly, instead of simply reporting the performance of debugging, we set our major focus as "to identify the reasons and give explanations to the observed phenomena". Mooctest recorded all of the operations of participants during their debugging process, which were used for tracking and analyzing their focus of attention. From the analysis, navigation patterns in debugging with SBFL and the impact from SBFL on fault detection were investigated. We formulated the following three research questions:

- RQ1: Can Mooctest benefit debugging? Specifically, does the accuracy of fault localization affect debugging efficiency?
- RQ2: Is there any particular navigation pattern on source code during debugging with SBFL? If yes, how does it affect the efficiency and does it have any relation with the accuracy of fault localization ? By addressing this question, we want to find reasons for the observations in RQ1.
- RQ3: Does the assistance from SBFL have any impact on programmers' fault detection abilities?

4. EXPERIMENT

4.1 Mooctest - An on-line examination platform with SBFL

We have developed Mooctest [19], an on-line examination platform for a Coursera course on software testing, to provide all supportive features. The subcomponent in Mooctest specifically for SBFL is called MDebug, which consists of three major modules:

(1) **Fault Localizer** provides static source code analysis for control flow graph generation, dynamic coverage analysis for execution profile construction, and risk analysis for suspiciousness calculation. We borrowed Jones et al.'s [12] idea to colorize the code according to their suspiciousness without changing their positions, such that the risk can be pinpointed while the integrity and readability of the program can still be preserved. Different from [12], we only colorized high-risk statements as red, but distinguished their suspiciousness with different levels of saturation: the higher the level, the greater the suspiciousness. As suggested by [21] that programmers are normally interested in only several top-ranked statements, Fault Localizer highlights the statements ranked as the top four or tied for the top four. The formula used to calculate the suspiciousness can be configured. In our experiment, Ochiai [1] was adopted.

(2) **Operation Tracker** records each operation during debugging, e.g. file opening, cursor movement, mouse clicking, file modification, and JUnit test execution, with a time stamp, such that we can track a programmer's operations to obtain more details about his task completion.

(3) **Log Analyzer** receives the log file and performs various analyses to provide in-depth knowledge about the SBFL-assisted debugging process.

The server of Mooctest assigned debugging tasks with an associated JUnit test suite to the participants. It also specified whether the SBFL feature was switched on. Participants were required to login with their Mooctest accounts and run the MDebug client on their local machines to perform the assigned tasks. During debugging, Operation Tracker recorded all the necessary information in the logs and sent them back to the server for marking and analyzing. Figure 1 illustrates the Mooctest GUI for a debugging task with the SBFL feature switched on, which is associated with the testing results of the provided test suite.

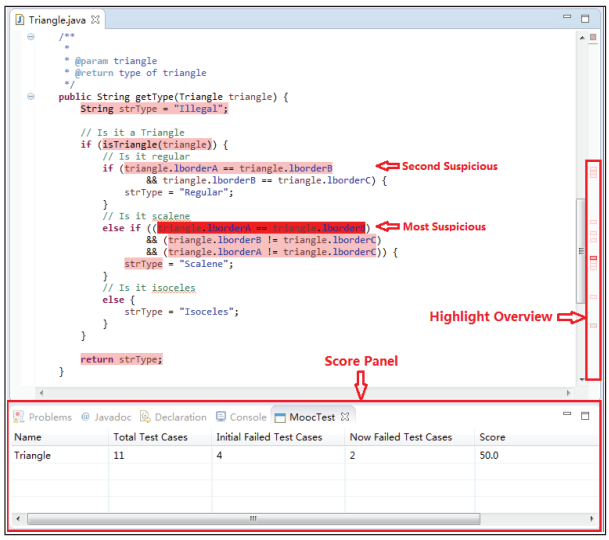


Figure 1: The SBFL feature on Mooctest

4.2 Participants and program subjects

There were 207 participants from two classes involved in our experiment where Class I and II had 97 and 110 participants, respectively. They were 3rd-year college students in Computer Science from our university, who were enrolled in our Coursera course on software testing, a compulsory on-line subject. These students participated in our experiment during an examination, which required them to concentrate and deliver more reliable results than volunteers. Besides, they all had about two years of Java programming experience. Their programming skills were diverse, which can help to reduce the bias in our analysis.

In the experiment, we selected seven programs from classic Computer Science algorithms, whose brief introduction is given in Table 1. As compared with the two programs used by Parnin and Orso [21], ours were smaller in size, but should have a more suitable level of difficulty for our student participants: comprehending these programs that have nested loops as well as complex data structure and logic is non-trivial, but is still accomplishable. Besides, these algorithms involved no domain expertise, which allowed our analysis to focus on the impact from MDebug and code comprehension.

Table 1: Program subjects

Program	Description	LOC	Type
QuickSort	Quick sorting algorithm	35	S
HeapSort	Heap sorting algorithm	35	S
FindTriple	Find all triples with sum=0 in an integer array	38	S
SubSequence	Get the number of all subsequences of a string	17	S
TicTacToe	Noughts&crosses game	500	M
Elevator	An elevator scheduler	238	M
BoatBomb	A game of guessing the "boat" in an $n \times n$ grid	164	M

†S means single-file program and M represents multiple-file program.

As shown in Table 1, there are four single-file and three multiple-file programs, which partially indicate two levels of complexity in debugging. For each program, we artificially seeded exactly one fault on one code line to get a faulty version. We obtained 17 faulty versions for all these program subjects. As compared with the two faulty programs in [21] where each one was associated with only

one failed test case in descriptive way, each of our faulty programs was associated with a set of executable JUnit test cases. Testing results were different on these 17 versions and hence the SBFL results varied as well. In contrast, the plug-in in [21] hard-coded the SBFL ranking results in a static XML file.

In order to provide a comprehensive analysis on the impact from SBFL accuracy, we categorized these 17 versions according to their SBFL results. First, we used "hit" to indicate whether the faulty statement s_f is red with the highest saturation: $hit = 1$ if yes; and 0 otherwise. For those cases where $hit = 1$, "$precision$" was then used to further distinguish the results with many false positives from those with few. In our context, $precision = \frac{1}{1+N_c}$, where N_c is the number of correct statements in red with the same highest saturation as s_f. The higher the $precision$, the better result provided by SBFL. For the cases where $hit = 0$, i.e. s_f does not have red with the highest saturation, we considered the distance in terms of the number of code lines, and data dependency between s_f and its nearest line in red with the highest saturation.

Finally, we categorized the 17 faulty versions into six groups, as shown in Table 2. And the category of each faulty version is shown in Table 3, which also gives the number of all test cases (2^{nd} column) and the number of the failed test cases (3^{rd} column).

Table 2: Categorization strategy

$hit = 1$		$hit = 0$			
$p \geq \alpha$	$p < \alpha$	$dist \leq \beta$ && $dep = 1$	$dist \leq \beta$ && $dep = 0$	$dist > \beta$ && $dep = 1$	$dist > \beta$ && $dep = 0$
A	B	C	D	E	F

†p means precision, $dist$ represents code line distance, and dep indicates whether there is data dependency (1 means yes, and 0 means no). In our experiment, α=$1/8$ and β=$LOC * 0.1$.

Table 3: All faulty versions

Faulty version	Test cases	Failures	Category
QuickSort_a	30	21	E
QuickSort_b	30	15	E
HeapSort_b	30	10	C
HeapSort_c	30	15	E
FindTriple_a	12	6	E
FindTriple_b	12	8	A
FindTriple_c	12	3	A
SubSequence_a	39	22	A
SubSequence_b	39	20	A
TicTacToe_a	38	24	F
TicTacToe_b	38	21	D
TicTacToe_c	38	29	D
Elevator_a	40	9	D
Elevator_b	40	20	E
BoatBomb_a	38	21	C
BoatBomb_b	38	13	B
BoatBomb_c	38	17	B

4.3 Experimental procedure

A tutorial on SBFL theory and the MDebug tool were first given to all participants to make sure that they fully understood the information provided by the tool. Then, these two classes attended an examination. Participants from each class were required to perform two debugging tasks within 90 minutes, where one was on a single-file program and the other one was on a multiple-file program. As a reminder, participants were told that they would be scored according to whether they fixed the bugs and passed all given test cases, rather than their time consumption. The timeframe of 90 minutes was imposed to avoid endless debugging. Class I was required to debug the single-file program by hand but to debug the multiple-file program with SBFL. Class II had the reversed set-up. The tasks were randomly assigned to each participant. Table 4 summarizes the experiment design.

Table 4: Experiment design

	Class I	Class II
Single-file program	without SBFL	with SBFL
Multiple-file program	with SBFL	without SBFL

5. RESULTS AND ANALYSIS

In this section, we will present our empirical results to address the three research questions[4].

5.1 RQ1: Can Mooctest benefit debugging?

To address this question, we analyzed the task completion rate and completion time to measure the effectiveness and the efficiency of debugging, respectively. For each debugging task, i.e. to repair one program, with or without the SBFL assistance, completion rate R_c is defined as S_c/S_{all}, where S_c is the number of participants who have successfully fixed the bug within the required timeframe; and S_{all} is the number of all participants involved in this task. For each individual participant who has correctly fixed the bug within the required timeframe, completion time T_c represents the time cost for successfully repairing the program.

(1) **Completion rate** R_c. Intuitively, we were expecting that with the SBFL assistance, Groups A and B, where the faulty statement has been successfully pinpointed in red with the highest saturation, should have increased R_c. Since Group A has even better accuracy than Group B, the improvement of Group A should be greater than that of Group B. For Groups C, D, E and F, the improvement of R_c, if there is any, should be smaller than those of Groups A and B.

Figure 2 demonstrates this comparison for each group, where the results for Groups A and B comply with our expectation that both of them have obvious improvement on R_c when SBFL is switched on. R_c in Groups A and B increase by 14.8% and 11.8%, respectively, which is also consistent with the intuition. On the other hand, for Groups C, D, E and F which do not correctly pinpoint the fault, the assistance from SBFL hardly shows any advantage: Groups E and F demonstrate similar R_c of debugging with and without SBFL; and Groups C and D even have the R_c decrease with SBFL. Although we cannot foresee this result before the experiment, the decreased R_c is still understandable; it is very likely that the bad SBFL results have side-tracked programmers, and hence brought in noise during the process. Concerning the comparison among Groups C, D, E and F, there is no obvious relationship observed. Groups C and D do not show any advantage over Groups E and F, which indicates that a small distance between the fault and its nearest highlighted statement seems not helpful to mitigate the difficulties in debugging with inaccurate fault localization results. By viewing the comparison results of "Group C v.s. Group D" and "Group E v.s. Group F", data dependency between the faulty and the highlighted statements tends to be not useful either.

In summary, it is reasonable to conclude that under certain conditions, the SBFL assistance in Mooctest does help to improve the completion rate. One dominant factor is the "hit" value of

[4]All the raw data and analysis are available on http://mooctest.net/academic/mdebugICSE16.

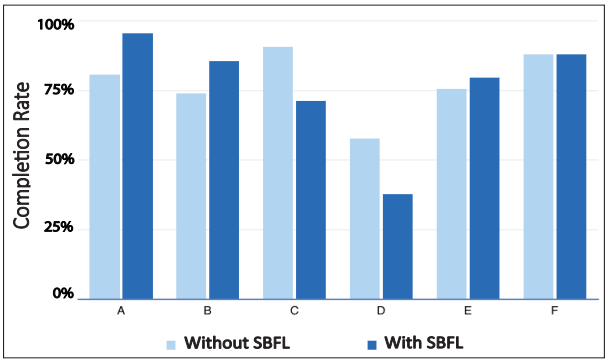
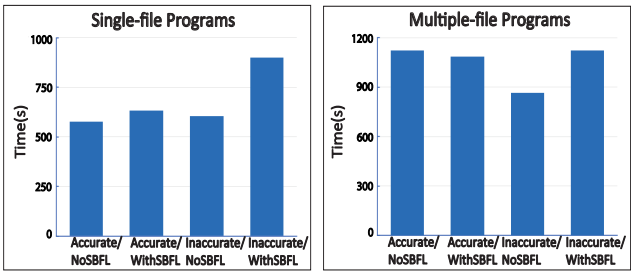


Figure 2: Completion rate for each group of faulty programs

the SBFL result: only when the faulty statement is correctly pinpointed (i.e. ranked or tied at the top of the list), does R_c tend to increase with the SBFL assistance. The low precision (i.e. many statements are tied at the top) may slightly weaken this advantage; while the small line distance and data dependency may not be able to compensate for the impact from the inaccurate localization results. These observations verify the explanation for "no impact from the accuracy" and the feasibility of the suggested "absolute ranking" in [21]. Therefore, in the following discussion, the six groups of faulty versions will be merged into two, namely Accurate Group G_{acc} where $hit = 1$ (including Groups A and B), and Inaccurate Group G_{in} where $hit = 0$ (including Groups C, D, E and F).

(2) **Completion time** T_c. In terms of the completion time, we want to investigate whether Mooctest can be helpful in decreasing the time cost to successfully fix a bug.

Figure 3(a) demonstrates the comparison of average T_c values. The first and second bars are for G_{acc} without and with SBFL, respectively, where the average T_c with SBFL (631.9 sec) is slightly longer than that without SBFL (578.2 sec). The third and fourth bars are for G_{in} without and with the SBFL feature, respectively, where the average T_c increases by 48.5% from the cases without SBFL (606.2 sec) to the cases with it (900.3 sec). If we compare the helpfulness of the SBFL feature between G_{acc} and G_{in} (shown in the second and the fourth bars, respectively), it can be found that the average T_c of G_{in} is about 42.3% higher than that of G_{acc}. However the average time cost of these two groups is actually very similar (shown in the first and the third bars) when debugging by hand. These results imply that SBFL prolongs the debugging process, especially in inaccurate cases.



(a) single-file (b) multiple-file

Figure 3: Average completion time T_c

Similar results can be concluded from the comparison among multiple-file programs in Figure 3(b). For G_{acc}, the SBFL feature helps to slightly reduce the average T_c (the first and the second bars). Regarding G_{in}, similar to the above results for the single-file programs, the average T_c significantly increases by 30.1% with the SBFL feature (the third and the fourth bars). Our 2-sample T-tests (both 2-tailed and 1-tailed) shown in Tables 5 and 6 further verify the above comparison[5].

Table 5: T-test on T_c for single-file programs

X	Y	H_0	H_1	p-value	Accept
G_{acc} NoSBFL	G_{acc} SBFL	$X=Y$	$X\neq Y$	0.63204	H_0
G_{in} NoSBFL	G_{in} SBFL	$X\geq Y$	$X<Y$	0.00543	H_1
G_{acc} SBFL	G_{in} SBFL	$X\geq Y$	$X<Y$	0.01423	H_1

Table 6: T-test on T_c for multiple-file programs

X	Y	H_0	H_1	p-value	Accept
G_{acc} NoSBFL	G_{acc} SBFL	$X=Y$	$X\neq Y$	0.86358	H_0
G_{in} NoSBFL	G_{in} SBFL	$X\geq Y$	$X<Y$	0.0362	H_1
G_{acc} SBFL	G_{in} SBFL	$X=Y$	$X\neq Y$	0.81623	H_0

In summary, we can conclude that generally speaking, SBFL may not be helpful in reducing the time cost of debugging, even for those with accurate localization results. Concerning the cases which give inaccurate results, SBFL may even lead to significant time increase. These observations are surprisingly interesting and somehow counter to our intuition. At first, we conjectured that the longer average T_c may be caused by the increased R_c which implies that some uncompleted debugging tasks become resolved with SBFL, and these tasks normally require higher time cost. However, by viewing the detailed data, we reckoned that the involved high time cost from these difficult tasks should not be the major contribution to the increase of average T_c because (1) actually there were only two groups of programs having increased R_c, while for the other four groups whose R_c decreased, we still observed, even more clearly, increased T_c; (2) in all groups including A and B, not just the top high T_c with SBFL were higher than those without it, the cases of low T_c had a similar tendency. Therefore, in order to find out the explanations, we further investigated the following research questions.

5.2 RQ2: Is there any navigation pattern in SBFL-assisted debugging?

In order to gain insight into what was happening during the SBFL-assisted debugging process, and find the reason for the above observations, we tracked and analyzed participants' focus of attention, from which two navigation patterns were observed, as follows.

(1) ***First scan*** pattern. As introduced in Section 2.1, Uwano et al. [33] identified a "scan" pattern in static code review, where they found that the duration of "scan" presented a positive correlation with the efficiency of review. This observation is reasonable since a careful "scan" shall give the reviewer a better understanding about the code and hence will be helpful in identifying the fault more quickly.

Motivated by [33], we are curious as to whether there also exists a similar scan pattern in debugging activities with SBFL.

[5]All hypothesis tests in this paper is set to be under the significant level of 0.05.

Different from [33] in which eye-movement was used to track reviewers' focus of attention, we used participants' operations on the screen such as cursor movement, mouse clicking, file opening, and code modification to achieve this goal. In fact, neither eye movements nor these operations can be 100% precise to reflect a programmer's focus of attention. However, as shown in both Psychology and Computer Human Interaction studies [5, 8, 11], operations such as cursor and mouse movements can be eligible surrogates with good practicality.

Figures 4 and 5 give two sample navigation patterns on the source code without and with SBFL, respectively, where the horizontal axis records the elapsed time and the vertical axis indicates the code line. The red dot line (in Figure 5 only) and the blue dashed lines (in both Figures 4 and 5) that run through the whole debugging process represent the statements colored as red with the highest saturation (presented to programmers with the SBFL feature) and the actual faulty locations (hidden to programmers during the process), respectively. The black line segments record the duration of the programmer's focus on the corresponding code lines. And the orange points on some black line segments indicate modifications on these lines by the programmer.

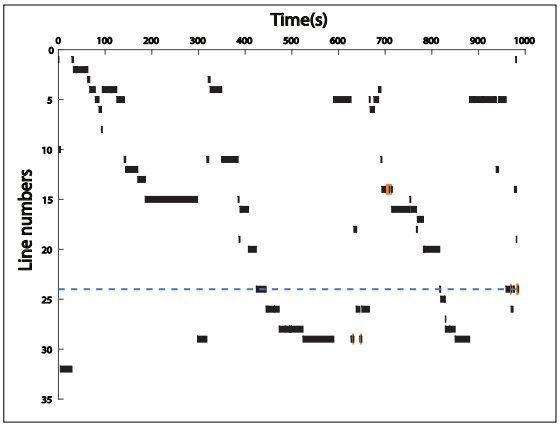


Figure 4: Code navigation without SBFL

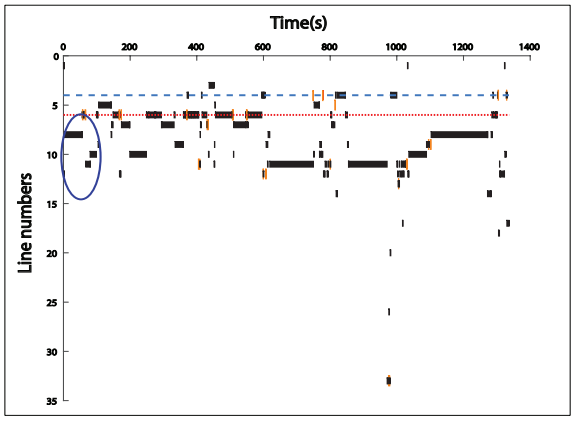


Figure 5: Code navigation with SBFL

Figure 4 illustrates that debugging without SBFL has a similar pattern as static code review in which programmers normally have a first scan followed by several rounds of navigation along through all the code lines sequentially and finally reach a particular area and fix the bug. For debugging with SBFL, shown in Figure 5, the code navigation presents a different pattern, i.e., programmers no longer iteratively follow the natural sequential order of the code lines. Instead, they frequently switch their focus of attention between the red lines with the highest saturation and the others. Interestingly, we have found that the "first scan" pattern seems to be preserved in debugging with SBFL, although it becomes less obvious. In the theory of SBFL, programmers are expected to inspect the code lines along the given ranking list. However, we found that regardless of the accuracy of the SBFL results, there was almost nobody who directly jumped into the red line with the highest saturation at their first visit on the code. Instead, a "first scan" was performed (which may not be on the entire code), then was followed by the inspection of the red line with the highest saturation. However, the duration of this "first scan" varied across different participants: some were longer while others could be very short such that only a few lines were skimmed within a short period of time, as the one in Figure 5, circled in blue.

This observation is not surprising. However, we are more interested in whether the duration of this "first scan" (denoted as T_1) affects final debugging efficiency, in the similar way that "scan" affects static code review [33].

Different from [33], we did not investigate the relationship between the absolute T_1 and T_c (the completion time). Instead, we defined $R_1{=}T_1/T_c$ to evaluate the portion of the effort that a programmer put in his "first scan" out of his total effort in fixing the bug. We wanted to investigate the impact of R_1 on T_c for debugging with SBFL. We defined, for SBFL-assisted debugging, cases where $R_1{<=}10\%$ as "short scan" while cases where $R_1{>}10\%$ as "long scan". As a consequence, all participants' debugging with SBFL can be split into two parts, namely, cases with "short scan" and with "long scan".

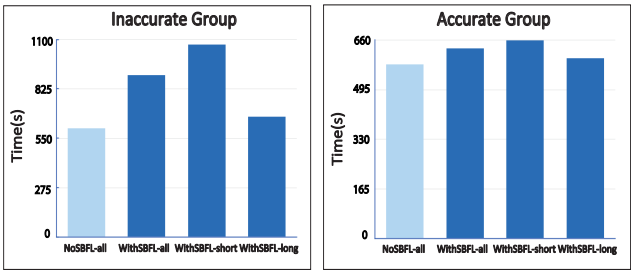
For single-file programs belonging to the inaccurate group G_{in}, T-tests comparing T_c among debugging with "long scan", "short scan" and "no SBFL assistance", are presented in Table 7.

Table 7: T-test on T_c for different R_1 (G_{in})

X	Y	H_0	H_1	p-value	Accept
SBFL Long	SBFL Short	$X{\geq}Y$	$X{<}Y$	0.01563	H_1
NoSBFL	SBFL Short	$X{\geq}Y$	$X{<}Y$	0.00204	H_1
SBFL Long	NoSBFL	$X{=}Y$	$X{\neq}Y$	0.61299	H_0

Table 7 shows that T_c with "short scan" tends to be significantly larger than those with "long scan", as well as those without SBFL; while "long scan" and debugging without SBFL produce no significant difference on their T_c. This implies that among all the T_c of SBFL-assisted cases, smaller ones (i.e. cases with higher debugging efficiency) tend to appear with "long scan"; while higher ones (i.e. cases with lower efficiency) usually belong to debugging with "short scan". The increase on T_c with SBFL in G_{in} is very likely due to the existence of many "short scan" cases. On the other hand, "long scan" could be helpful in reducing the gap between debugging with and without SBFL.

Figure 6(a) depicts the consistent results on the average T_c in a more vivid manner, where the leftmost bar is for debugging without SBFL, and the second, third and fourth bars are for the SBFL-assisted debugging in all cases, "short scan" and "long scan", respectively. The average T_c for debugging with "short scan" is about 59.9% and 76.9% higher than those for debugging with "long scan" and without SBFL, respectively.



(a) G_{in} (b) G_{acc}

Figure 6: Average T_c with different R_1

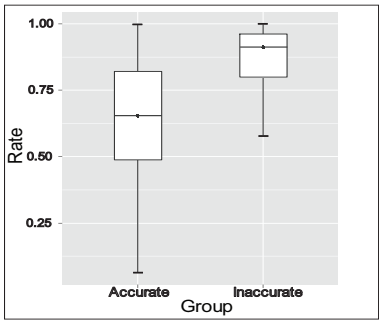


Figure 7: R_2 in the follow-up browsing

And for G_{acc}, 2-tailed T-tests did not indicate any significant difference. The p-values for the same comparisons as Rows 1, 2, 3 in Table 7 become 0.84481, 0.6651, and 0.58687, respectively. However the average T_c comparison shown in Figure 6(b) reveals some slight difference, which is similar to G_{in} but less obvious. It can be found that for G_{acc}, the average T_c for "short scan" (the third bar) is about 10.0% and 13.9% higher than those for the "long scan" (the fourth bar) and "debugging without SBFL" (the leftmost bar), respectively.

Similar results can also be found in multiple-file programs. In summary, quickly focusing on the highlighted lines without sufficient first code scan could be harmful to the efficiency of debugging. And the situation could be even worse when the SBFL results are inaccurate. This observation is consistent with the one in [33]. This is most likely because regardless of SBFL-assisted debugging or manual static code review, the success of fault detection shall always eventually rely on human intelligence. Reading and comprehending the source code are inevitable to finally detect and correct the faults. A more careful scan on the code will generally lead to a better understanding of the program, and hence a more efficient debugging.

This observation makes us re-examine the original motivation of SBFL, one of whose goals is to help programmers quickly enter the most suspicious code area when given a program to debug. However, as discussed above, the debugging process may not be accelerated in this way. This will be further discussed in Section 6.

(2) ***Follow-up browsing*** pattern. As shown in Figure 5, from the first visit on the red lines with the highest saturation to the completion of debugging, participants tend to switch their focus of attention between the highlighted lines and the others. However, we also found that they actually spent most of the time outside rather than on the red lines with the highest saturation. Here, we use single-file programs as illustrations.

Let T_2 denote the total time that a programmer spends on statements outside the red lines with the highest saturation after his first scan, and R_2 denote $T_2/(T_c - T_1)$. Figure 7 gives the boxplot diagrams for the distribution of R_2 in G_{acc} and G_{in}. In G_{in} (the right box) about 98.0% of the participants have an R_2 over 50%, and 85.7% of the participants even have an R_2 higher than 70%. The average R_2 is as high as 84.8%. For G_{acc} in the left box, the R_2 values are relatively lower. However, 75% of the participants still have an R_2 over 50%. The average R_2 for the accurate group is 63.9%.

In other words, no matter whether or not the SBFL result is accurate, programmers normally tend to spend most of their rest debugging time (after the first scan) outside the red lines with the highest saturation. Inaccurate SBFL results normally lead to more time cost outside the highlighted lines. This phenomenon implies that no matter how precisely we pinpoint the buggy lines, "wasting time" on a wider range of the source code is generally indispensable, and even fundamental, since it is helpful in comprehending the context. Obviously, simply relying on the disjoint colored code lines cannot be helpful to collect such information.

This raises doubts about another goal of SBFL that is to pinpoint the most suspicious statements and make programmers concentrate on these risky lines without wasting time on other "safe" lines. Based on our observation, even if we assumed that the lines outside the pinpointed ones were really safe, there was still a large portion of time spent on these safe areas. Then, does it mean that highlighting the risky code to save inspection efforts on those "safe" statements is infeasible during practical debugging activities? Again, we will discuss this in Section 6.

So far, it seems that we have found some possible explanations to the decrease in debugging efficiency with SBFL. In summary, no matter whether programmers claim that they are reluctant to follow the SBFL results, the colorization will affect their debugging behaviors, more or less. Some may trust the SBFL result very much and would like to quickly start to check the pinpointed lines; while others may not feel comfortable to check these lines until they have a longer scan and better understanding on the code. Generally speaking, the former cases have shown an obvious harmfulness to the efficiency of debugging, especially when provided with inaccurate SBFL results. After the first visit on the pinpointed code lines, programmers will switch their focus of attention between these lines and others. Most of their time has actually been spent outside these lines, in order to comprehend the context and hence to detect and correct the fault. As a consequence, the critical factor that reduces the helpfulness of SBFL may not be the ones that we normally suspected, such as the accuracy and reliability of SBFL results, or people's being reluctant to follow the localization results. The critical factor is most likely residing in the interference between the mechanism of automated fault localization and the actual assistance needed by programmers in debugging. Based on these observations, we now delve further to investigate the impact from SBFL on fault comprehension and identification.

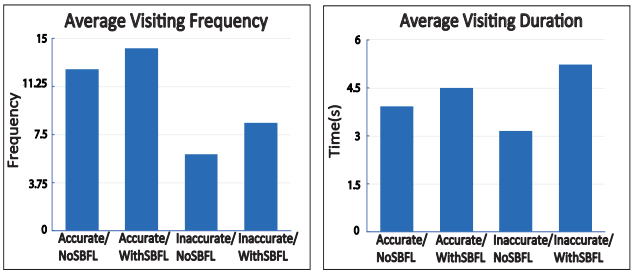
5.3 RQ3: Will SBFL affect fault detection?

It has been widely accepted by the SBFL research community that there is no perfect bug detection. Simple inspection of a fault is not sufficient for immediately detecting it (i.e. understand and identify the fault). However, there is a lack of evidence about whether SBFL has any impact on fault detection. Thus in this study we analyzed participants' operation logs to address this question.

For both debugging with and without SBFL, we counted the frequency of each participant's visits on the real faulty line without

detecting it (denoted as V_f), and the average duration of all his visits on the faulty line (denoted as D_f).

Figure 8(a) demonstrates the average V_f for debugging on programs of G_{acc} and G_{in}, without and with SBFL, which confirms that regardless of the debugging method, there is generally no "perfect bug detection". For the same debugging task, debugging with SBFL tends to have a higher frequency: for programs in G_{acc} the average V_f with SBFL (the second bar) is 13.2% higher than that of debugging by hand (the first bar). When the SBFL is inaccurate (G_{in}), this difference is as high as 40.5% (the third and the fourth bars for debugging without and with SBFL, respectively). These imply that when using SBFL, programmers tend to revisit the bug more times in order to detect it.



(a) Average V_f (b) Average D_f

Figure 8: Fault revisit

A similar tendency can be found on D_f, shown in Figure 8(b). For G_{acc}, the average D_f tends to be slightly longer when using SBFL (4.5 sec, in the second bar) than by hand (3.9 sec, in the first bar). Similarly, for G_{in}, the average D_f with SBFL (the fourth bar) is about 2 sec longer than that without it (3.2 sec in the third bar).

The above comparison results are consistent with the one in RQ1 that debugging with SBFL is slower than debugging by hand. With these results, we now have more understanding about the average ability that people detect a fault, and have learnt that the fault revisit frequency and the average duration on these visits may increase when the SBFL feature is switched on. These phenomena appear to be more obvious when the SBFL result is inaccurate. It is very likely because SBFL may confuse programmers in their code comprehension and slightly weaken their bug detection abilities.

5.4 Threats to validity

First, we have used programmers' operations such as cursor movement, mouse clicking, file opening and changing, to represent their focus of attention on the code. However, this information may not always be precisely reflecting the real situation. Another way to achieve this goal is to use eye-movement. However, as shown in previous studies [5, 8, 11], both eye-movement and operations contain noise, and the latter one can be eligible surrogates with good practicality. Actually, the operations may contain less noise than eye-movement in our context. Based on our observation, participants may randomly shift their eye-attention to any place within or even outside the screen, but not really for the purpose of inspecting the code. For example, they may frequently check the time, especially when the 90 minutes were about to run out. On the other hand, we observed that our participants always moved their cursor along each code line that they were inspecting. Therefore, we feel that in our experiment, tracking the operations may provide the data more closely approximate to the fact.

Secondly, although we recruited many more participants in our experiment than compared to prior work, these participants were all students with very similar education background. There were no professional programmers with industrial experience involved. However, our debugging tasks were neither from industry projects nor involved any domain knowledge. As a consequence, these students should be eligible representatives in our experiment [25]. Actually, students may even provide more reliable results than volunteers from the industry because students participated in our experiment in the form of an examination, which required them to concentrate. Besides, these students from two classes have presented diverse Java programming skills, which can help to reduce the bias in our analysis.

The final threat to validity is in regards to the representativeness of the program subjects, which were small-size programs with seeded faults. Actually, as discussed in Section 4.2, these programs should be suitable for our participants. Besides, we have confirmed that the seeded faults did not include those "too obvious or stupid", such as $a = a + b$ being mutated to $a = a$. More importantly, these programs did not involve any domain knowledge or advanced programming skills, thus could be proper to investigate the impact from an SBFL tool and code comprehension. As a reminder, though we adopted both single-file and multiple-file programs to represent two levels of difficulty in debugging, we actually did not observe much difference between these two types of programs. Thus, in our future studies, we will explore programs with more levels of debugging difficulty, to investigate how different levels of difficulty affect the results.

6. INSIGHTS AND SUGGESTIONS

6.1 Actual debugging process

In Section 5, RQ1 first reveals a decrease in the efficiency of debugging with SBFL. By investigating RQ2 and RQ3, we have found some possible explanations. On one hand, by quickly entering the area of the suggested suspicious code, a programmer may miss a good global understanding about the software. He may have to waste more time to compensate for being unfamiliar with the source code. Concerning the cases where the suggestion is actually misleading, even more extra efforts may be required. In contrast, a good "first scan", which has been regarded as an efficient pattern for programmers in static code review [33], turns to be also essential in SBFL-assisted debugging. On the other hand, continuously and exclusively focusing on the pinpointed code lines is generally unrealistic. Browsing the context is more necessary in detecting the fault. Moreover, suggestions from SBFL may interfere with programmers fault detection.

These observations conflict with the motivation of automated fault localization, which helps localize the most suspicious code and save the time that programmers waste on the irrelevant code. It is generally believed that debugging involves fault localization (denoted as J_A), as well as code comprehension and fault correction (denoted as J_B) [21]. The total cost of debugging is decided by the costs of these two tasks. Automated fault localization aims to reduce the time cost and human involvement in J_A by providing programmers with some suggested fault locations, and expects that such saving can lead to the final improvement on the total debugging efficiency. However, this expectation has an assumption that J_A and J_B are independent and performed sequentially in debugging, such that saving on J_A by SBFL can lead to saving on the total cost. But in practice, these two tasks are tightly coupled and there is no clear boundary between them. Thus, it could be very likely that saving on J_A has some impact on J_B, such that final debugging efficiency is not changed as expected. In the following two subsections, we will discuss some possible solutions to the conflict between the motivation of automated debugging and the real-life situation.

6.2 Different fault localization techniques for different levels of debugging

Although there are some opinions that statistical fault localization (such as SBFL) may be helpful in small-scale programs, but not suitable for large-scale systems where the small percentage of the to-be-inspected code will result in a large number of statements, we are holding a different view.

Based on the above analysis in Section 5, we conjecture that such automated debugging techniques are not actually helpful to small-scale programs, where the fault localization is always performed on statement level. In contrast, these techniques may be useful in large-scale systems, however, not for the purpose of locating the **faulty statement**, but the **faulty module**. These suggestions are not only because a coarser granularity can significantly reduce the absolute number of the to-be-inspected entities for large-scale systems, but are also supported by the following evidences.

(1) Fault may not be localizable in statement level. Thung et al. [32] observed that many real-life faults span across multiple code lines; some even involve multiple methods. This fact makes locating a real-life fault very difficult. On the other hand, we observed that it is unlikely that one fault could span across multiple modules. In other words, it is more feasible to locate a faulty module than to locate a faulty statement.

(2) In locating the faulty statement for a unit program, due to the nature of the control-dependency, it is impossible to always rank the faulty statement at top. Distributions of e_p (the number of passed test cases that cover the statement) and e_f (the number of failed test cases that cover the statement) could be very different in the entry statement, a loop predicate, a branch predicate, an assignment statement and the exit statement, and such difference is not controllable by test cases but due to the nature of their roles in the control flow graph. Any of these statements could be faulty, thus a general method cannot guarantee to pinpoint the fault all the time. In contrast, the topological graph in high-level of a large system could be different, especially those following the principles of high-cohesion and low-coupling. There are fewer tight control-dependencies. The distribution of e_p and e_f on these modules are relatively easier to control. Hence there is much higher chance to make an accurate diagnose [4].

(3) Due to the high-cohesion, a module is usually self-descriptive. For example, it is very easy to understand the major functions of Class Postman in a PostOffice system, without learning too much about other classes, especially when there are well-written JavaDoc. But given a statement like "for (int i = 0; i < N; i++)", nobody can comprehend it immediately. As a consequence, programmers can quickly focus on the pinpointed module to start identifying the fault. The time saved by the fault localization technique has much higher chance to result in a total saving on the efficiency.

Therefore, we propose to first adopt statistical fault localization on large-scale systems to isolate the suspicious module. Then, for each suspicious module, a more sophisticated method like program slicing is suggested, which is not only more accurate, but also informative for code comprehension.

6.3 Improving the practicality

Based on the above discussion, we now summarize our suggestions in improving the practicality of automated fault localization.

First, apply the right technique on the right scenario. As suggested above, to achieve a better debugging performance, a smart combination of various fault localization techniques is necessary.

Secondly, accuracy still matters. It is always important not to mislead programmers. This can be done by designing better algorithms, performing program refactoring, or adjusting test suites [39]. Some researchers also proposed to predicate the accuracy before really utilizing the localization results [17].

Thirdly, incorporate effective code comprehension assistance. Integration of automated debugging and code comprehension assistance shall be helpful to provide better practicality.

Finally, build a platform that can provide the features and information really needed by programmers in different scenarios.

7. CONCLUSIONS AND FUTURE WORKS

Although not many studies in automated debugging have taken human factors into account, it is undeniable that human involvement plays a critical role in this activity. In this study, we revisited the helpfulness of Spectrum-Based Fault Localization, by examining programmers' actual performance in debugging with its assistance. Our experiment involving 207 participants and 17 debugging tasks indicated that an accurate localization result was helpful in terms of completing the debugging task. However, regardless of the accuracy, SBFL was not helpful in terms of improving the efficiency of debugging. And an inaccurate SBFL result may lead to an even longer debugging process. To identify the reason, we tracked and analyzed programmers' focus of attention, from which source code navigation patterns (namely, "first scan" and "follow-up browsing") and SBFL's impact on bug detection were identified. We found that (1) a slow and careful "first scan" that provides a better understanding about the code is favorable for a quicker debugging process; (2) browsing on the context of the pinpointed code is essential; and (3) a natural sequential code scan tends to be more efficient than an SBFL-guided one, in detecting a fault.

These observations reconfirmed the importance of code comprehension in automated debugging. More importantly, they indicated that the decrease in debugging efficiency was most likely due to the interference between the mechanism of automated fault localization and the actual assistance needed by programmers in debugging. To resolve the interference, we proposed a conjecture that it is not the problem of the technique itself. Instead, it should be due to an inappropriate application of the technique. And we further gave some suggestions on how to properly perform automated debugging, in order to have an actual improvement on the efficiency.

In our future studies, we will focus on improving the practicality of automated debugging from the suggested directions, such as the applicability of a technique, an integration of automated debugging and code comprehension assistance, a better debugging platform, etc. And their actual helpfulness to programmers will be examined.

Acknowledgment

This work is partially supported by the National Key Basic Research and Development Program of China (973 Program 2014CB340702) and the National Natural Science Foundation of China (Grant No. 61572375, 61170067, 61373013, 91418202, 61472178).

8. REFERENCES

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, pages 39–46, Riverside, USA, 2006.

[2] R. Bednarik. Expertise-dependent visual attention strategies develop over time during debugging with multiple code

representations. *International Journal of Human-Computer Studies*, 70(2):143–155, 2012.

[3] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif. The impact of identifier style on effort and comprehension. *Empirical Software Engineering*, 18(2):219–276, 2013.

[4] C. Chen, H.-G. Gross, and A. Zaidman. Improving service diagnosis through increased monitoring granularity. In *Proceedings of the 7th IEEE International Conference on Software Security and Reliability*, pages 129–138, Gaithersburg, Maryland, USA, 2013.

[5] M. Chen and V. Lim. Eye gaze and mouse cursor relationship in a debugging task. In *Proceedings of HCI International Posters' Extended Abstracts*, pages 468–472. Springer, 2013.

[6] M. E. Crosby and J. Stelovsky. How do we read algorithms? a case study. *Computer*, 23(1):25–35, 1990.

[7] M. A. Francel and S. Rugaber. The value of slicing while debugging. *Science of Computer Programming*, 40(2):151–169, 2001.

[8] J. Freeman, R. Dale, and T. Farmer. Hand in motion reveals mind in motion. *Frontiers in Psychology*, 2:59, 2011.

[9] J. D. Gannon. Human factors in software engineering. *Computer*, 12(12):6–7, 1979.

[10] C. Gouveia, J. Campos, and R. Abreu. Using html5 visualizations in software fault localization. In *Proceedings of the First IEEE Working Conference on Software Visualization*, pages 1–10, Eindhoven, Netherlands, 2013.

[11] J. Huang, R. W. White, and S. Dumais. No clicks, no problem: Using cursor movements to understand and improve search. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1225–1234, Vancouver, BC, Canada, 2011.

[12] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, Florida, USA, 2002.

[13] A. Ko, B. Myers, M. Coblenz, and H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, 2006.

[14] A. J. Ko and B. A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 151–158, Vienna, Austria, 2004.

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

[16] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. Fleming. How programmers debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering*, 39(2):197–215, 2013.

[17] T.-D. B. Le, D. LO, and F. Thung. Should i follow this fault localization tool's output? automated prediction of fault localization effectiveness. *Empirical Software Engineering*, pages 1–38, 2015.

[18] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead. Does bug prediction support human developers? findings from a google case study. In *Proceedings of 35th IEEE/ACM International Conference on Software Engineering*, pages 372–381, San Francisco, USA, 2013.

[19] Mooctest. http://mooctest.net/.

[20] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology*, 20(3):11:1–11:32, 2011.

[21] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 199–209, Toronto, Canada, 2011.

[22] D. Piorkowski, S. Fleming, C. Scaffidi, L. John, C. Bogart, B. John, M. Burnett, and R. Bellamy. Modeling programmer navigation: A head-to-head empirical evaluation of predictive models. In *Proceedings of 2011 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 109–116, Pittsburgh, PA, USA, 2011.

[23] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th International Conference on Software Engineering*, pages 390–401, Hyderabad, India, 2014.

[24] R. Saha, M. Lease, S. Khurshid, and D. Perry. Improving bug localization using structured information retrieval. In *Proceedings of the 28th International Conference on Automated Software Engineering*, pages 345–355, Palo Alto, California, USA, 2013.

[25] I. Salman, A. T. Misirli, and N. Juristo. Are students representatives of professionals in software engineering experiments? In *Proceedings of 37th IEEE/ACM International Conference on Software Engineering*, pages 666–676, Florence, Italy, 2015.

[26] B. Sharif, M. Falcone, and J. I. Maletic. An eye-tracking study on the role of scan time in finding source code defects. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, pages 381–384, Santa Barbara, California, 2012.

[27] B. Sharif, J. Maletic, et al. The effects of layout on detecting the role of design patterns. In *Proceedings of the IEEE Conference on Software Engineering Education and Training*, pages 41–48, Pittsburgh, USA, 2010.

[28] J. Silva and O. Chitil. Combining algorithmic debugging and program slicing. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 157–166, Venice, Italy, 2006.

[29] I. Sommerville. *Software Engineering*. Pearson, 9 edition, 2009.

[30] R. Stein and S. E. Brennan. Another person's eye gaze as a cue in solving programming problems. In *Proceedings of the 6th international conference on Multimodal interfaces*, pages 9–15, State College, USA, 2004.

[31] Y. Tao, J. Kim, S. Kim, and C. Xu. Automatically generated patches as debugging aids: a human study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 64–74, Hong Kong, China, 2014.

[32] F. Thung, D. Lo, L. Jiang, et al. Are faults localizable? In *Proceedings of the IEEE 9th Working Conference on Mining Software Repositories*, pages 74–77, Zurich, Switzerland, 2012.

[33] H. Uwano, M. Nakamura, A. Monden, and K.-i. Matsumoto. Analyzing individual performance of source code review

using reviewers' eye movement. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, pages 133–140, San Diego, USA, 2006.

[34] 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*, pages 1–11, Baltimore, Maryland, USA, 2015.

[35] M. Weiser and J. Lyle. Experiments on slicing-based debugging aids. In *Proceedings of the Workshop on Empirical Studies of Programmers*, pages 187– 197, USA, 1986.

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

[37] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology*, 22(4):31:1–31:40, 2013.

[38] J. Xuan and M. Monperrus. Learning to combine multiple ranking metrics for fault localization. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pages 191–200, Victoria, Canada, 2014.

[39] J. Xuan and M. Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 52–63, Hong Kong, China, 2014.

[40] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering*, pages 319–329, Portland, Oregon, 2003.