IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received August 22, 2020, accepted September 17, 2020, date of publication September 21, 2020,
date of current version September 30, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3025460

Improving Software Fault Localization by Combining Spectrum and Mutation

ZHANQI CUI[1], (Member, IEEE), MINGHUA JIA[1,2], (Student Member, IEEE), XIANG CHEN[3], (Member, IEEE), LIWEI ZHENG[1], AND XIULEI LIU[1]
[1]Computer School, Beijing Information Science and Technology University, Beijing 100101, China
[2]School of Information, Central University of Finance and Economics, Beijing 100081, China
[3]School of Information Science and Technology, Nantong University, Nantong 226019, China

Corresponding author: Xiulei Liu (liuxiulei@bistu.edu.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2017YFB1400402, in part by the National Natural Science Foundation of China under Grant 61702041 and Grant 61601039, and in part by the Qin Xin Talents Cultivation Program of Beijing Information Science Technology University under Grant QXTCP C201906 and Grant QXTCP B201905.

ABSTRACT The performance of software fault localization techniques is critical to software debugging and the reliability of software. Spectrum-based fault localization (SBFL) and mutation-based fault localization (MBFL) are the two most popular fault localization methods. However, the accuracies of the two methods are still limited. For example, only 10.63% of faults can be detected by inspecting the top 3 suspicious elements reported by Ochiai, which is a famous SBFL technique. Unfortunately, programmers only examine the first few suspicious elements before losing patience. Since the information used in SBFL and MBFL are quite different and complementary, this paper proposes a novel approach by combining spectrum and mutation to improve the fault localization accuracy. First, the faulty program is evaluated by using SBFL, and the potential faulty statements are ranked according to their suspiciousness. Then, mutants of the program are generated and executed by MBFL. Finally, the statements that are ranked in the top tied *n* by SBFL are evaluated and reranked according to their mutation scores. Experiments are carried on the Defects4J benchmark and the results reveal that the accuracy of the proposed approach outperforms those of the SBFL and MBFL techniques. In terms of the faults located by inspecting the top 1 suspicious elements, the SMFL techniques detect at least 2.36 times more faults than two SBFL techniques (DStar and Ochiai) and detect at least 1.86 times more faults than two MBFL techniques (MUSE and Metallaxis).

INDEX TERMS Software debugging, fault localization, program spectrum, mutation testing.

I. INTRODUCTION

As software becomes more complex and error prone, software testing and fault localization become critical to software quality. Fault localization techniques, which play important roles in program analysis and software testing, focus on efficiently identifying faulty program elements that cause software failures. In the whole life cycle of software, the debugging process is the most expensive and time-consuming stage, especially for fault localization activities. As the scale and complexity of software increase, increasingly more faults could exist in software, and testing software is one of the keys to improving software quality. The failures detected by testing need to be fixed by debugging. Typically, fault localization is the most expensive part of debugging.

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

Fault localization techniques focus on how to use the static or dynamic information of software to accurately and quickly locate program elements that cause failures. The existing fault localization techniques can mainly be classified into seven types, including spectrum-based fault localization (SBFL) [1]–[7], mutation-based fault localization (MBFL) [8]–[11], dynamic program slicing [12]–[14], stack trace analysis [15], [16], prediction switching [17], information retrieval-based fault localization [18], and history-based fault localization [19], [20]. Among these seven types of fault localization techniques, SBFL and MBFL are the most popular fault localization methods.

SBFL uses the program spectrum to localize faults. The program spectrum is a collection of program information that provides a specific matrix on the dynamic behaviors of software [1], [2]. SBFL is based on spectrum coverage information and test case execution results. Typically, an element is suspicious if it is executed multiple times by failed test cases but is seldom executed by passed test cases. However, many suspicious elements are frequently ranked the same by SBFL. For instance, the elements in a basic block are all executed or not executed by different test cases; hence, the elements in a basic block are always ranked the same [1], [2]. In addition, the accuracy of SBFL is limited, especially for the top suspicious elements. In [2], SBFL techniques successfully localized only 4.5% of faults on average by inspecting the top 1 suspicious element, which means that only 45 out of 1000 programs that contain faulty elements can be successfully localized by inspecting the top 1 suspicious element of the ranked lists.

MBFL is based on mutants, where a mutant typically changes one statement by replacing one operand or expression with another [2], [8], [9], [21]. It calculates the mutation score of each mutant and averages the mutation scores of the mutants of an element as the suspicious value of the element. Intuitively, an element is suspicious if it affects failed test cases more frequently and affects passed test cases more rarely. However, the accuracy of MBFL is still limited, even though it costs a huge amount of time to generate and execute mutants. In [2], MBFL techniques successfully localized 6.5% of faults on average by inspecting the top 1 suspicious element, which means that 65 out of 1000 faulty programs can be successfully localized by inspecting the top 1 element of the ranked lists.

However, previous studies suggested that programmers will only inspect a few positions at the top of a ranked list before losing patience [10], [22]. In [22], programmers exhibited some form of jumping between positions when confirming the faults of programs according to a ranked list. 37% of the jumps skipped 10 positions on average. However, 95% of the reviews start with the top 1 suspicious element in the ranked list, which means that the top 1 position is very rarely skipped by programmers. Therefore, how to improve the fault localization accuracy, especially for the top 1 suspicious element, is crucial to the practicability of fault localization. We notice that SBFL and MBFL methods utilize different information of spectrum and mutation, respectively. The previous research also found that the two methods are weakly correlated [2]. This finding suggests that it is possible to improve the fault localization accuracy by combining the spectrum and mutation.

A. PROPOSED SOLUTION

In this paper, we propose a novel approach that combines Spectrum and Mutation for Fault Localization (SMFL). First, the faulty program is evaluated by using SBFL based on the spectrum of the coverage information and execution results of test cases, and the potential faulty statements are ranked according to their suspiciousness. Then, mutants of the program are generated and executed by MBFL. Finally, the statements that are ranked as top tied n by SBFL are reranked according to their mutation scores. The approach leverages the advantages of both SBFL and MBFL to improve the fault localization accuracy. Specifically, four SMFL-based techniques, including OMuse (combines Ochiai [4] and MUSE [8]), OMetallaxis (combines Ochiai and Metallaxis [9]), DMuse (combines DStar [11] and MUSE) and DMetallaxis (combines DStar and Metallaxis), are presented. To further validate the effectiveness of the SMFL approach, experiments are carried on the Defects4J[1] benchmark and the results reveal that the accuracy of the proposed method significantly outperforms those of the spectrum-based and mutation-based techniques.

B. CONTRIBUTIONS

The main contributions of this paper are summarized as follows:

- a novel SMFL fault localization approach, which improves the fault localization accuracy and efficiency.
- four SMFL-based techniques, including OMuse, OMetallaxis, DMuse and DMetallaxis, are proposed for fault localization.
- an experimental evaluation of SMFL in comparison with SBFL and MBFL on the Defects4J benchmark.

The rest of the paper is organized as follows. Section II describes the SMFL approach. Section III presents the experimental design and evaluation. Section IV reviews related works. Finally, the conclusion and the discussion of the future work are provided in section V.

II. OUR APPROACH

The SMFL approach is based on combining spectrum-based fault localization and mutation-based fault localization. The flowchart of the approach is presented in Figure 1. First, the faulty program is evaluated by using SBFL based on spectrum coverage information and test case execution results, and the potential faulty statements are ranked according to their suspiciousness. Then, mutants of the program are generated and executed by MBFL. Finally, the statements ranked as top tied n by SBFL are selected as candidates, and the candidates are reranked according to their mutation scores. The following part of this section introduces the SMFL fault localization approach in details.

A. EVALUATE THE PROGRAM BY USING SBFL

SBFL uses the program spectrum and test results to localize software faults. The program spectrum is a collection of coverage information and execution results of test cases. The spectrum of a program, which usually is in the form of a matrix, describes the dynamic behaviors of the program. The testing results record whether a test case has failed or passed. The more frequently an element is executed by failed tests, and the less frequently it is executed by passed tests, the more suspicious the element [1], [2]. Conversely, the less frequently an element is executed by failed tests, and the more frequently it is executed by passed tests, the less suspicious the element.

[1]Defects4J: http://fault-localization.cs.washington.edu/

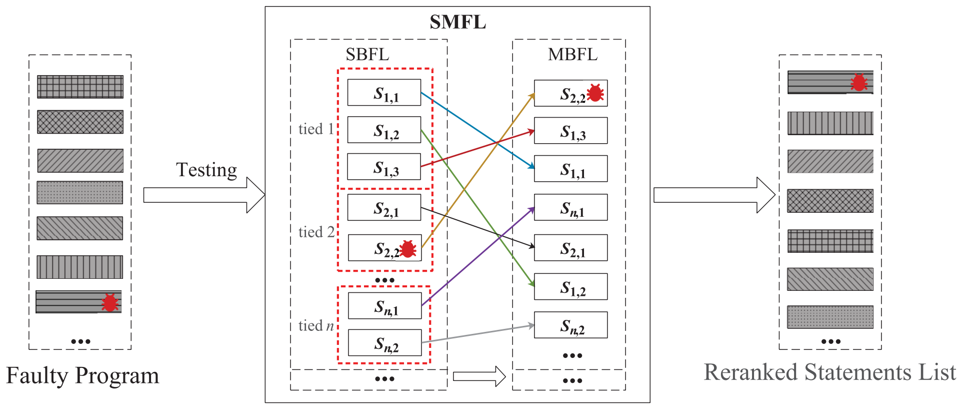


FIGURE 1. The flow chart of the SMFL approach.

For different SBFL techniques, the suspiciousnesses of elements are calculated by different suspiciousness formulas with the same spectrum of the program and test cases. As a result, different SBFL techniques are strongly correlated. Ochiai and DStar are the SBFL techniques with the best performance on real bugs in [21]. In [2], Ochiai and DStar, which are selected as the representative SBFL techniques, can respectively locate approximately 44% and 43% of the faults by inspecting the top 10 suspicious elements. In [23], Ochiai was selected as a representative coefficient-based technique and DStar archived good results in many cases. Therefore, we choose the Ochiai and DStar techniques as the representative SBFL techniques to evaluate the suspiciousness of statements in this step.

For a faulty program, the execution spectrum is constituted by a binary matrix *Ma* and a test result vector *Re*. *Ma* records the coverage information of the executing test cases for all statements. *Re* records the testing results of each test case. For a statement s_i, the execution information of the statement can be extracted from the program spectrum and represented as a 3-tuple $(e_{i,f}, e_{i,p}, n_{i,f})$, where $e_{i,f}$ indicates the number of failed test cases that execute s_i, $e_{i,p}$ indicates the number of passed test cases that execute s_i, and $n_{i,f}$ indicates the number of failed test cases that not execute s_i. Figure 2, in which each row gives the execution information of a test case, is an example of a program spectrum. In columns 2-8, '1' represents that the corresponding statement is executed by the test case while '0' represents that the corresponding statement is not executed by the test case. The last column, in which '+' indicates pass and '−' indicates fail, represents the results of each test case. A 3-tuple (1, 3, 0) represents the spectrum information of s_1, which means that s_1 is executed by one failed test case and three passed test cases. A 3-tuple (0, 2, 1) represents the spectrum information of s_3, which means that s_3 is executed by two failed test cases; besides, there is another failed test case that does not execute s_3.

Equations (1) and (2) are the suspiciousness formulas used by Ochiai [4] and DStar [11], in which $Ochiai(s_i)$ and $DStar(s_i)$ represent the suspiciousness values of a statement

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	
tc_1	1	1	0	0	1	1	1	+
tc_2	1	1	1	1	0	0	1	+
tc_3	1	1	1	1	1	1	1	+
tc_4	1	1	0	0	0	0	1	-

FIGURE 2. An example of a program spectrum.

s_i. '*' in equation (2) is a variable, which we set to 2, as in the empirical study of Zou et al [2]. By using Ochiai or DStar, the potential faulty statements can be ranked according to their suspiciousness values. The suspiciousness values of some statements could be the same. Especially, statements in a basic block are always executed or not executed by different test cases, and the suspiciousness values of the statements in one basic block are all the same. We rank the statements of the program according to their suspiciousness values calculated by Ochiai or DStar to a sequence $S = \langle Tied_1, Tied_2, \ldots, Tied_i, \ldots \rangle$. $Tied_i$ is a subsequence of S, which is composed of statements $\langle s_{i,1}, s_{i,2}, s_{i,j}, \ldots \rangle$ with the same suspiciousness value and ranked as tied i. The statements in $Tied_i$ are ranked by the line number of the statements in ascending order.

$$Ochiai(s_i) = \frac{e_{i,\,f}}{\sqrt{(e_{i,\,f} + n_{i,\,f}) \cdot (e_{i,\,f} + e_{i,\,p})}} \tag{1}$$

$$DStar(s_i) = \frac{e^{*}_{i,\,f}}{e_{i,\,p} + n_{i,\,f}}, \quad (* = 2, 2.5, 3, \ldots) \tag{2}$$

B. EVALUATE THE PROGRAM BY USING MBFL

MBFL techniques are based on the execution information of mutants, where a mutant typically changes one statement by replacing one operand or expression with another [2], [8], [9]. Mutation operators can be categorized as statements, operations, variables, constants, etc. Typically, an element is suspicious if it affects failed test cases more frequently and affects passed test cases more rarely. In contrast, if an element affects passed test cases more frequently and affects failed test cases more rarely, the element is less suspicious.

As a previous study shown, MUSE and Metallaxis are two of the MBFL techniques with the best performance [2]. In [2], MUSE and Metallaxis can respectively locate approximately 19% and 36% of the faults by inspecting the top 10 suspicious elements. Equation (3) and Equation (4), as shown at the bottom of the next page are the formulas used by MUSE and Metallaxis, respectively. The performance of MUSE and Metallaxis are quite different because the two techniques extract different information on a mutation test, and different factors are used. Moreover, a previous study found that these two MBFL techniques are weakly correlated [2]. Thus, we evaluate the program using both the MUSE and Metallaxis techniques to get different mutation scores of the statements, respectively.

Table 1 shows the 9 mutation operators, which are used to generate mutants of programs [24]. The descriptions and examples of the mutation operators are given in the table. Let us suppose that $Mu = \{M_1, M_2, \ldots, M_i, \ldots\}$ is a set of mutant sets for the statements of a program P, where $M_i = \{m_{i,1}, m_{i,2}, \ldots\}$ is a set of mutants generated for a statement s_i. Most of the statements have multiple mutants but there are still some statements having no mutants because these statements cannot be mutated. The mutation score of the mutants are calculated using mutation testing. For a statement s_i, T_i is the subset of the original test cases T that executed s_i. The factors used in MUSE and Metallaxis include *totalfailed*, *failed*$(m_{i,j})$, *passed*$(m_{i,j})$, *f2p*, and *p2f*. *totalfailed* indicates the number of failed test cases for the original program P. For the execution result of a mutant $m_{i,j}$, *failed*$(m_{i,j})$ and *passed*$(m_{i,j})$ are used to indicate the number of test cases that failed on P but now pass on mutant $m_{i,j}$ and the number of test cases that passed on P but now fail on a mutant $m_{i,j}$, respectively. *f2p*, and *p2f* are used to indicate the number of test cases with results that change from failure to pass on any mutant and the number of test cases with results that change from pass to failure on any mutant, respectively. In Equation (3), the factors of the MUSE suspiciousness formula contain *f2p*, *p2f*, *failed*$(m_{i,j})$, and *passed*$(m_{i,j})$; while in Equation (4), the factors of the Metallaxis suspicious formula contain *totalfailed*, *failed*$(m_{i,j})$, and *passed*$(m_{i,j})$.

C. COMBINE SPECTRUM AND MUTATION

The third step of SMFL is to rerank the candidates reported by SBFL according to their mutation scores. OMuse, OMetallaxis, DMuse, and DMetallaxis are proposed by combing spectrum and mutation for fault localization.

Algorithm 1 describes how to combine the spectrum and mutation for fault localization. In line 5, the sequence $S = \langle Tied_1, Tied_2, \ldots, Tied_i, \ldots\rangle$ is the result of an SBFL technique, such as Ochiai or DStar. In line 6, S is split into two subsequences $S' = \langle Tied_1, Tied_2, \ldots, Tied_n\rangle$ and $S'' = \langle Tied_{n+1}, Tied_{n+2}, \ldots\rangle$. In lines 8-10, a set of mutants

TABLE 1. Mutation operators.

Operator	Description	Example
AOR	Arithmetic Operator Replacement	a + b ↦ a - b
LOR	Logical Operator Replacement	a ^b ↦ a | b
COR	Conditional Operator Replacement	a || b ↦ a && b
ROR	Relational Operator Replacement	a == b ↦ a >= b
SOR	Shift Operator Replacement	a > > b ↦ a < < b
ORU	Operator Replacement Unary	-a ↦ ~a
EVR	Expression Value Replacement	return a ↦ return 0
		int a = b ↦ int a = 0
LVR	Literal Value Replacement	0 ↦ 1
		1 ↦ -1
		1 ↦ 0
		true ↦ false
		false ↦ true
		"Hello" ↦ ""
STD	STatement Deletion	return a ↦ <no-op>
		break ↦ <no-op>
		continue ↦ <no-op>
		foo(a,b) ↦ <no-op>
		a = b ↦ <no-op>
		++a ↦ <no-op>
		–a ↦ <no-op>

are generated for each statement in P. In line 12, mutation testing is carried out on the generated mutants. In lines 13-20, the mutation score of the statements in P' are calculated by mutation formulas, such as MUSE and Metallaxis. In line 21, the statements in P' are reranked according to their mutation scores to generate a suspicious subsequence S'''. For the statements in S'', the original order reported by SBFL is kept. S''' is spliced with S'' together into a reranked suspicious statements list S.

In this paper, OMuse, OMetallaxis, DMuse, and DMetallaxis are proposed by combing the spectrum and mutation for fault localization. OMuse is based on the combination of Ochiai and MUSE while OMetallaxis is based on the combination of Ochiai and Metallaxis. DMuse is based on the combination of DStar and MUSE while DMetallaxis is based on the combination of DStar and Metallaxis. It should be noted that in OMetallaxis and DMetallaxis, only the statements in P' need to be mutated in lines 8-10. This is because the variables *f2p* and *p2f*, which are related to all the mutants of P, are not used in Equation (4).

III. EXPERIMENTS AND EVALUATIONS

A. EXPERIMENTAL DESIGN

We implemented OMuse, OMetallaxis, DMuse, and DMetallaxis based on the proposed SMFL approach, which combines Ochiai, DStar, MUSE, and Metallaxis, respectively. In our empirical study, we select the statements ranked $Tied_1$ (n=1) as candidates for reranking to evaluate our approach. To evaluate the effectiveness of the approach, the research intends to answer the following research questions:

RQ1: Is it common for multiple elements to have the same suspiciousness value by SBFL?

RQ2: How effective is SMFL compared with SBFL?

RQ3: How effective is SMFL compared with MBFL?

Algorithm 1 Combine the Spectrum and Mutation for Fault Localization

```
Input:
    Program $P$, spectrum $Sp$, mutants $Mu$, test cases $T$
Output:
    Reranked suspicious statements list: $S$
1:  for $s_i$ in $P$ do
2:      get $e_{i,f}$, $e_{i,p}$, $n_{i,f}$ from $Sp$
3:      get suspiciousness value of $s_i$ by $Ochiai(s_i)$
4:  end for
5:  $S$ = rank $s_i$ in $P$ according to $Ochiai(s_i)$
6:  split $S$ into $S' = \langle Tied_1, Tied_2, \ldots, Tied_n \rangle$ and $S'' = \langle Tied_{n+1}, Tied_{n+2}, \ldots \rangle$
7:  $P' = \{s_i \mid s_i \text{ in } S'\}$
8:  for $s_i$ in $P$ do
9:      $M_i \leftarrow$ a set of the mutants generated for $s_i$
10: end for
11: $Mu = M_1 \cup M_2 \cup \cdots \cup M_i \cup \cdots$
12: mutation test is carried out on $Mu$ with test cases $T$
13: get the value of the parameters, such as $totalfailed$, $f2p$, and $p2f$ based on the results of the mutation test
14: for $s_i$ in $P'$ do
15:     $M_i = \{m_{i,1}, m_{i,2}, \ldots\}$
16:     for $m_{i,j}$ in $M_i$ do
17:         get $failed(m_{i,j})$ and $passed(m_{i,j})$ based on the results of the mutation test
18:     end for
19:     get mutation score of $s_i$ by via MBFL
20: end for
21: $S'''$ = rerank statements in $S'$ according to their mutation scores
22: $S = \langle S'''', S'' \rangle$
23: return $S$
```

B. EXPERIMENTAL SUBJECTS

To answer the research questions, experiments are carried out on the Defects4J benchmark. Defects4J is a real-world fault dataset composed of 6 open-source Java projects. It is a collection of reproducible faults and a supporting infrastructure with the goal of advancing software engineering research. The Defects4J benchmark has been used as the experimental subject in many previous fault localization studies [2], [24]–[29]. The details of the Defects4J benchmark are summarized in Table 2. It contains 395 faults from the 6 open-source projects: Chart (26 faults), Closure (133 faults), Lang (65 faults), Math (106 faults), Mockito (38 faults), and Time (27 faults). For each fault, it provides a faulty version of the project. In Table 2, the first column is the names of the projects. The second column is the numbers of faults for the 6 projects. The third column is the average source lines of code (LOC) for each project. The fourth column is the number of average test cases for each project, which include at least one failed test case per fault. For each fault, Defects4J provides faulty and fixed program versions with a minimized change that represents the isolated bug fix. This change indicates which lines in a program are defective.

TABLE 2. Details of the Defects4J Dataset(V1.1.0).

Projects	Faults	LOC	Number of test cases
Chart	26	7057	428
Closure	133	30660	351
Lang	65	1731	88
Math	106	7036	305
Mockito	38	4252	964
Time	27	3959	738
Total	395	48748	2874

C. MEASUREMENT METRICS

Many measurements metrics are used to evaluate the accuracy and effectiveness of software fault localization techniques. In this paper, we adopt *EXAM*, $E_{inspect}@n$ and propose $E_{inspect}@Tied_n$ as the metrics to measure the performance of the SBFL, MBFL, and SMFL techniques.

EXAM is a commonly used metric for fault localization techniques to evaluate the average rank of faulty elements [2], [30]–[32]. The value of *EXAM* is the percentage of elements that must be inspected until a real faulty element is found [2], [10], [23], [26]. The smaller the value of EXAM is, the more effective the fault localization technique.

In their empirical research, Zou *et al.* [2] used $E_{inspect}@n$ to evaluate the absolute rank of the faulty elements. $E_{inspect}@n$ counts the total number of faults that were successfully localized by inspecting the top n positions of the ranked lists [2], [8], [22]. The bigger the value of $E_{inspect}@n$ is with a smaller n, the more effective the fault localization technique. It is worth noting that programmers pay more attention to the top suspicious elements in practice. As a pervious study shown [19], instead of checking the statements in a ranked list one by one until the hypothesis about the cause of the failure is confirmed, all the programmers exhibit some form of jumping between positions in a ranked list. Moreover, the top 1 position in a ranked list is very rarely skipped by

$$MUSE(s_i) = \frac{\sum_{m_{i,j} \in M_i} \left(failed(m_{i,j}) - \frac{f2p}{p2f} \cdot passed(m_{i,j}) \right)}{|M_i|}, \quad (M_i \in mutants(s_i)) \tag{3}$$

$$Metallaxis(s_i) = \frac{\sum_{m_{i,j} \in M_i} \left(\frac{failed(m_{i,j})}{\sqrt{totalfailed \cdot (failed(m_{i,j}) + passed(m_{i,j}))}} \right)}{|M_i|}, \quad (M_i \in mutants(s_i)) \tag{4}$$

TABLE 3. The distribution of different versions that are ranked as tied 1 by Ochiai.

Projects	Tied 1		Avg Size
	Only 1 element	More than 1 element	
Chart	6	20	122
Closure	40	93	21
Lang	18	47	10
Math	21	85	18
Mockito	20	18	5
Time	13	14	4
Total	118	277	22

TABLE 4. The distribution of different versions that are ranked as tied 1 by DStar.

Projects	Tied 1		Avg Size
	Only 1 element	More than 1 element	
Chart	4	22	116
Closure	40	93	9
Lang	21	44	16
Math	27	79	28
Mockito	23	15	4
Time	11	16	19
Total	126	269	22

the programmers. Therefore, $E_{inspect}@1$ is very important to evaluate the effectiveness of fault localization techniques.

In addition, we find that many statements are ranked as tied with the same suspiciousness values. To describe this situation in a more precise way, we propose $E_{inspect}@Tied_n$ as a metric. $E_{inspect}@Tied_n$ represents the number of the faults that can be successfully localized by inspecting the statements of the suspicious list ranked as tied 1 to tied n. With this metric, we can not only pay attention to the absolute positions of the faulty elements in the suspiciousness list using $E_{inspect}@n$, but we can also address the tied ranks of faulty elements using $E_{inspect}@Tied_n$.

D. EXPERIMENTAL RESULTS AND ANALYSIS

1) STATEMENTS WITH SAME SUSPICIOUS VALUES BY SBFL

The number of different versions of the projects, in which only 1 statement and more than 1 statement are ranked as tied 1 by Ochiai and DStar, are shown in Tables 3 and 4, respectively. As Tables 3 and 4 show, 277 (70%) and 269 (68%) defective programs that have more than 1 element are ranked as tied 1 by Ochiai and DStar, respectively. In Tables 3 and 4, except Mockito, more than 1 element is ranked as tied 1 in more than half of the different versions of the other 5 projects (Chart, Closure, Lang, Math, and Time). The column 'Avg Size' gives the average number of statements that are ranked as tied 1 in different versions of the projects. As Tables 3 and 4 shows, the average size of tied 1 is 22 for the 6 projects, which means that 22 statements on average are both ranked as tied 1 by Ochiar and DStar.

TABLE 5. The performance of Ochiai on $E_{inspect}@Tied_n$ (n = 1, 2, 3, 5).

Projects	$E_{inspect}$			
	$@Tied_1$	$@Tied_2$	$@Tied_3$	$@Tied_5$
Chart	11	12	15	16
Closure	10	21	27	39
Lang	18	28	32	37
Math	31	41	47	63
Mockito	7	7	11	14
Time	1	6	7	9
Total	78	115	139	178

TABLE 6. The performance of DStar on $E_{inspect}@Tied_n$ (n = 1, 2, 3, 5).

Projects	$E_{inspect}$			
	$@Tied_1$	$@Tied_2$	$@Tied_3$	$@Tied_5$
Chart	8	8	13	15
Closure	11	22	26	36
Lang	15	24	27	37
Math	25	34	39	57
Mockito	6	6	11	13
Time	5	6	7	9
Total	70	100	123	167

TABLE 7. The performance of Ochiai on $E_{inspect}@n$ (n = 1, 3, 5, 10).

Projects	$E_{inspect}$				$EXAM$
	@1	@3	@5	@10	
Chart	0	4	9	14	0.0616
Closure	2	6	18	31	0.0229
Lang	1	8	22	28	0.1455
Math	4	17	30	48	0.0613
Mockito	2	6	10	15	0.0683
Time	0	1	6	13	0.0599
Total	9	42	95	149	0.0330

Tables 5 and 6 describe the performance of Ochiai and DStar on $E_{inspect}@Tied_n$ ($n = 1, 2, 3, 5$). For example, $E_{inspect}@Tied_2$ indicates the number of faulty program versions that were successfully localized by inspecting the statements ranked as tied 1 and tied 2. The performances of Ochiai and DStar on $E_{inspect}@n$ ($n = 1, 3, 5, 10$) are listed in Tables 7 and 8, respectively. In Tables 7 and 8, we can find that if the programmers only check the first suspicious statement, only 9 and 11 out of 395 faults can be located by Ochiai and DStar, respectively. In comparison, 78 and 70 real faults can be found by inspecting the tied 1 statements that are ranked by Ochiai and DStar, respectively. If we can effectively rerank the tied 1 statements by using the information of the mutation test, up to 20% of the faults could be successfully located by only checking the top 1 statement.

Answer for RQ1: 70% and 68% of the defective programs have more than 1 element ranked as tied 1 by Ochiai and DStar, respectively, and the average number of statements ranked as tied 1 is 22 for both Ochiai and DStar.

TABLE 8. The performance of DStar on $E_{inspect}@n$ (n = 1, 3, 5, 10).

Projects	$E_{inspect}$				$EXAM$
	@1	@3	@5	@10	
Chart	0	1	4	9	0.0788
Closure	3	9	16	23	0.0128
Lang	1	5	10	19	0.1118
Math	4	10	12	28	0.0480
Mockito	2	5	9	12	0.0503
Time	1	2	5	12	0.0596
Total	11	32	56	103	0.0497

2) THE COMPARISON OF SMFL AND SBFL

To compare SMFL and SBFL, experiments are performed by using Ochiai, DStar, OMuse, OMetallaxis, DMuse, and DMetallaxis on Defects4J, respectively. To compare their performances, $E_{inspect}@n$ and $EXAM$ scores are used as the measurements metrics.

As Table 7 shows, Ochiai can locate 9, 42, 95, and 149 faults (approximately 2%, 11%, 24%, and 38% of all faults) by inspecting the top 1, 3, 5, and 10 suspicious statements, respectively. The performances of OMuse and OMetallaxis on $E_{inspect}@n$ (n = 1, 3, 5, 10) are listed in Table 9. As Table 9 shows, the performances of OMuse and OMetallaxis are almost the same expect for the small difference on $E_{inspect}@1$ for the Lang and Math projects. Both OMuse and OMetallaxis can locate 36, 60, 104, and 154 faults (approximately 9%, 15%, 26%, and 39% of all faults) by inspecting the top 1, 3, 5, and 10 suspicious statements, respectively, which are 4.00, 1.43, 1.09, and 1.03 times greater than those of Ochiai. The total $EXAM$ of Ochiai is 0.0330, and the total $EXAM$ of OMuse and OMetallaxis are both 0.0270. The improvement of $EXAM$ is only 0.0060 because only the statements ranked as tied 1 by SBFL are chosen to be reranked by SMFL, which accounts for just a small portion of the entire program.

As Table 8 shows, DStar can locate 11, 32, 56, and 103 faults (approximately 3%, 8%, 14%, and 26% of all faults) by inspecting the top 1, 3, 5, and 10 suspicious statements, respectively. Meanwhile, as shown in Table 10, DMuse can locate 26, 49, 66, and 112 faults (approximately 7%, 12%, 17%, and 28% of all faults) by inspecting the top 1, 3, 5, and 10 suspicious statements, respectively, which are 2.36, 1.84, 1.18, and 1.09 times greater than those of DStar. DMetallaxis can locate 27, 50, 68, and 113 faults (approximately 7%, 13%, 17%, and 29% of all faults) by inspecting the top 1, 3, 5, and 10 suspicious statements, respectively, which are 2.45, 1.56, 1.21, and 1.10 times greater than those of DStar. The total $EXAM$ of DStar is 0.0497, and the total $EXAM$ of DMuse (DMetallaxis) is 0.0444 (0.0443). The improvement of $EXAM$ is approximately 0.0050 because only the statements ranked as tied 1 by SBFL are chosen to be reranked by SMFL, which accounts for just a small portion of the entire program.

TABLE 9. The performance of OMuse and OMetallaxis on $E_{inspect}@n$ (n = 1, 3, 5, 10) and $EXAM$.

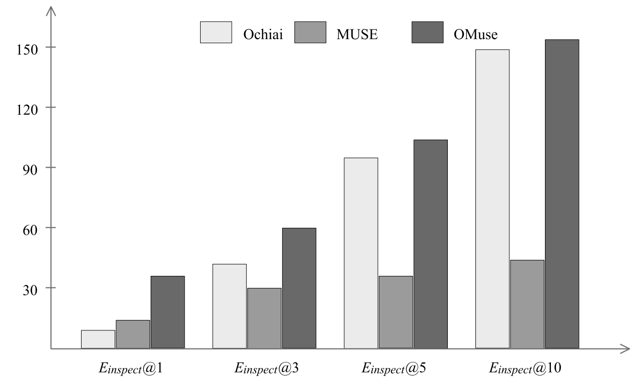
Projects	Technique	$E_{inspect}$				$EXAM$
		@1	@3	@5	@10	
Chart	OMuse	3	5	9	14	0.0488
	OMetallaxis	3	5	9	14	0.0488
Closure	OMuse	3	6	18	31	0.0266
	OMetallaxis	3	6	18	31	0.0266
Lang	OMuse	5	10	23	29	0.1455
	OMetallaxis	4	10	23	29	0.1455
Math	OMuse	16	25	35	50	0.0557
	OMetallaxis	17	25	35	50	0.0557
Mockito	OMuse	7	11	13	17	0.0675
	OMetallaxis	7	11	13	17	0.0675
Time	OMuse	2	3	6	13	0.0598
	OMetallaxis	2	3	6	13	0.0598
Total	OMuse	36	60	104	154	0.0270
	OMetallaxis	36	60	104	154	0.0270

TABLE 10. The performance of DMuse and DMetallaxis on $E_{inspect}@n$ (n = 1, 3, 5, 10) and $EXAM$.

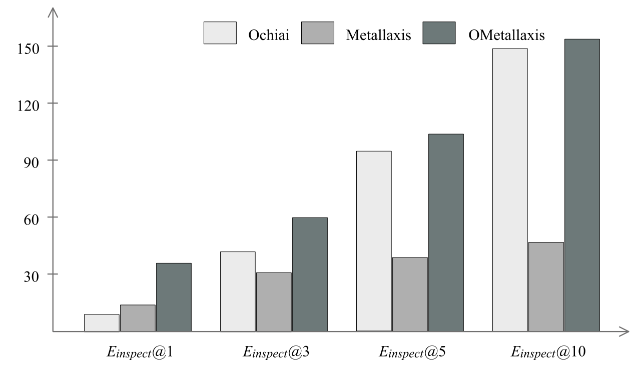
Projects	Technique	$E_{inspect}$				$EXAM$
		@1	@3	@5	@10	
Chart	DMuse	3	4	7	11	0.0710
	DMetallaxis	3	4	7	11	0.0710
Closure	DMuse	4	9	16	23	0.0128
	DMetallaxis	4	9	16	23	0.0128
Lang	DMuse	3	7	10	21	0.1116
	DMetallaxis	3	6	10	21	0.1116
Math	DMuse	8	15	16	30	0.0479
	DMetallaxis	9	17	18	31	0.0478
Mockito	DMuse	5	9	11	15	0.0011
	DMetallaxis	5	9	11	15	0.0011
Time	DMuse	3	5	6	12	0.0594
	DMetallaxis	3	5	6	12	0.0594
Total	DMuse	26	49	66	112	0.0444
	DMetallaxis	27	50	68	113	0.0443

The results of comparing SMFL with Ochiai and DStar by $E_{inspect}@n$ are also shown in Figure 3. In Figure 3, we can see that SMFL techniques (OMuse, DMuse, OMetallaxis and DMetallaxis) can find more faults than SBFL techniques (Ochiai and DStar) according to $E_{inspect}@1$, $E_{inspect}@3$, $E_{inspect}@5$, and $E_{inspect}@10$.

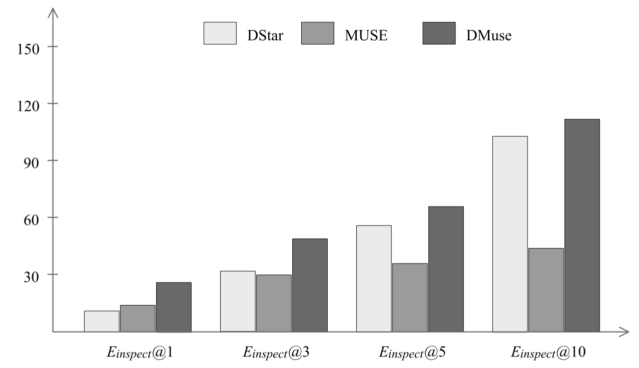
Answer for RQ2: The accuracies of the SMFL techniques are better than those of the SBFL techniques. Specifically, SMFL techniques can localize more faults than Ochiai and DStar, and the $EXAM$ of the SMFL techniques are also reduced.



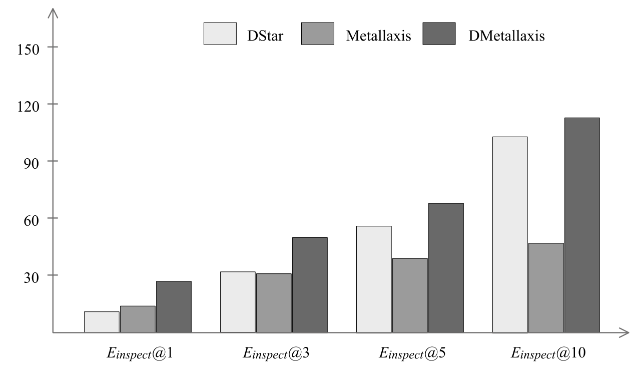
(a) Comparing OMuse with Ochiai and MUSE



(b) Comparing OMetallaxis with Ochiai and Metallaxis



(c) Comparing DMuse with DStar and MUSE



(d) Comparing DMetallaxis with DStar and Metallaxis

FIGURE 3. The results of comparing SMFL with SBFL and MBFL on $E_{inspect}@n$.

TABLE 11. The performance of MUSE and Metallaxis on $E_{inspect}@n$ ($n = 1, 3, 5, 10$) and *EXAM*.

Projects	Technique	$E_{inspect}$				$EXAM$
		@1	@3	@5	@10	
Chart	MUSE	0	1	1	3	0.4427
	Metallaxis	0	1	1	3	0.4427
Closure	MUSE	0	3	3	3	0.4887
	Metallaxis	0	3	3	3	0.4887
Lang	MUSE	4	9	9	9	0.4313
	OMetallaxis	4	9	9	9	0.4313
Math	MUSE	8	14	18	24	0.3872
	Metallaxis	8	15	21	27	0.3684
Mockito	MUSE	0	0	2	2	0.4740
	Metallaxis	0	0	2	2	0.4740
Time	MUSE	2	3	3	3	0.4451
	Metallaxis	2	3	3	3	0.4451
Total	MUSE	14	30	36	44	0.4142
	Metallaxis	14	31	39	47	0.4035

3) THE COMPARISON OF SMFL AND MBFL

Table 11 shows the performance of MUSE and Metallaxis, which are MBFL techniques, on $E_{inspect}@n$ ($n = 1, 3, 5, 10$). MUSE can locate 14, 30, 36, and 44 faults (approximately 4%, 8%, 9%, and 11% of all the faults) by inspecting the top 1, 3, 5, and 10 suspicious statements, respectively. Metallaxis can locate 14, 31, 39, and 47 faults (approximately 4%, 8%, 10%, and 12% of all the faults) by inspecting the top 1, 3, 5, and 10 suspicious statements, respectively.

As Table 9 and Table 11 show, the number of faults localized by OMuse is 2.57, 2.00, 2.89, and 3.50 times greater than that of MUSE on $E_{inspect}@1$, $E_{inspect}@3$, $E_{inspect}@5$,$E_{inspect}@10$, respectively. Moreover, the *EXAM* of OMuse is 0.3872 lower than that of MUSE. The number of faults localized by OMetallaxis is 2.57, 1.93, 2.67, and 3.28 times greater than that of Metallaxis on $E_{inspect}@1$, $E_{inspect}@3$, $E_{inspect}@5$,$E_{inspect}@10$, respectively. Moreover, the *EXAM* of OMetallaxis is 0.3765 lower than that of Metallaxis.

As Table 10 and Table 11 show, the number of faults localized by DMuse is 1.86, 1.63, 1.83, and 2.55 times greater than that of MUSE on $E_{inspect}@1$, $E_{inspect}@3$, $E_{inspect}@5$,$E_{inspect}@10$, respectively. Moreover, the *EXAM* of DMuse is 0.3698 lower than that of MUSE. The number of faults localized by DMetallaxis is 1.93, 1.61, 1.74, and 2.40 times greater than that of Metallaxis on $E_{inspect}@1$, $E_{inspect}@3$, $E_{inspect}@5$,$E_{inspect}@10$, respectively. Moreover, the *EXAM* of DMetallaxis is 0.3592 lower than that of Metallaxis.

The results of comparing SMFL with SBFL and MBFL by $E_{inspect}@n$ are also shown in Figure 3. In Figure 3, we can see that the SMFL techniques (OMuse, DMuse, OMetallaxis and

TABLE 12. Time costs of mutant generation and test execution (in seconds).

Projects	MUSE	Metallaxis	OMuse	OMetallaxis	DMuse	DMetallaxis
Chart	3757	3757	3757	1409	3757	1404
Closure	145170	145170	145170	1122	145170	746
Lang	308	308	308	3	308	4
Math	4852	4852	4852	42	4852	48
Mockito	39479	39479	39479	14	39479	574
Time	67775	67775	67775	96	67775	121
Total	261341	261341	261341	2986	261341	2897

DMetallaxis) can find more faults than the MBFL techniques (MUSE and Metallaxis) in terms of $E_{inspect}@1$, $E_{inspect}@3$, $E_{inspect}@5$,$E_{inspect}@10$.

As we know, MBFL techniques cost much more time than SBFL techniques because MBFL techniques need to generate a large number of mutants and the tests need to be executed multiple times. Therefore, we compare the time costs of the SMFL and MBFL techniques that are spent on mutant generation and test execution in table 12. As Table 12 shows, the time costs of OMuse, DMuse, MUSE, and Metallaxis are the same because all the mutants of the programs are generated and executed. Meanwhile, OMetallaxis and DMetallaxis only have approximately 11% of the time costs of the previous 4 techniques. The reason is that in equation (4), which is the suspiciousness equation of Metallaxis, the variables $f2p$ and $p2f$ contained in equation (3) are not included. To get the values of $f2p$ and $p2f$, all the mutants of the program need to be generated and executed. To get the values of the variables in equation (4), only the suspicious statements reported by SBFL need to be generated and executed. As a result, time costs are saved in OMetallaxis and DMetallaxis.

Answer for RQ3: The accuracy of the SMFL techniques outperforms that of the MBFL techniques. Specifically, SMFL techniques can localize more faults than MUSE and Metallaxis, and the *EXAM* of the SMFL techniques are also reduced. In addition, the OMetallaxis and DMetallaxis techniques save 89% of the time costs that are spent on mutant generation and test execution in comparison with the MBFL techniques.

E. THREATS TO VALIDITY

The threats to internal validity relate to the technique choices, experimental errors and biases. The choice of the SBFL and MBFL techniques in our study threatens the internal validity of the technique choices. We choose Ochiai, DStar, MUSE and Metallaxis because Ochiai and DStar are two of the SBFL techniques with the best performance, and MUSE and Metallaxis are two of the MBFL techniques with the best performance [2]. The effectiveness of the proposed SMFL method may result in different experiment results and conclusions when SMFL adopts other SBFL and MBFL techniques. The experiments are carried on the Defects4J benchmark dataset to mitigate the risk to the internal validity from experimental errors and biases. It can reduce experimental bias since Defects4J is a real-world fault dataset.

The threats to external validity relate to the generalizability of our proposed SMFL approach. The experiments are carried on a real-world fault dataset Defects4J. It contains 395 real faults from 6 open-source Java projects from real application scenarios. Each real fault is accompanied by a comprehensive test case that can expose that fault [27]. In the future, we will test more real programs in different programming languages with diverse faults using the SMFL method.

The threats to construct validity relate to the selection of the candidates for reranking. The statements ranked as tied 1 are chosen as candidates for reranking based on the mutation scores. The performance of SMFL may be improved by choosing statements ranked as tied 1 to tied n ($n > 1$) as candidates for reranking, but it needs to consider the complex relationships and weights between the suspiciousness values of the candidates reported by SBFL and the mutation scores reported by MBFL. We also realize that a different value of n might result in different experimental results and conclusions.

Another threat to the validity is that the experimental objects used are all single-fault programs. However, the SMFL approach can also be applied to programs with multiple faults. One simple solution is to produce fault-focused clusters to group failed test cases related to the same fault into the same clusters [33]. Then, the SMFL approach is applied to the successful test cases and the failed test cases in one cluster to locate the fault that is related to the cluster.

IV. RELATED WORK

Generally, fault localization identifies the most likely faulty elements via program analysis and the execution information of test cases. The program elements (code lines, statement blocks, functions, classes, etc.) are ranked and then checked one by one according to the order until the real fault is found [1], [2], [34]. SBFL and MBFL are two of the most popular fault localization methods. In this section, we briefly review the related work on spectrum-based fault localization and mutation-based fault localization.

A. SPECTRUM-BASED FAULT LOCALIZATION

SBFL uses the program spectrum to locate faults. Collofello and Cousins proposed that the program spectrum can be used for fault localization [34]. The program spectrum is a collection of program information that provides a specific matrix of the dynamic software behaviors [1], [2]. The specific matrix records the run-time profiles of the whole program statements for specific test cases. SBFL is based on spectrum coverage information and test case execution results. Typically, an element is suspicious if it is executed multiple times by failed test cases but it is seldom executed by passed test cases.

SBFL has received much research attention due to its simplicity and effectiveness. The effectiveness of risk evaluation formulas is an important research area of SBFL. However, the performance results of the risk evaluation formulas in SBFL strongly depend on the experimental setups, and the results are not comprehensive enough to provide a fair evaluation of investigated SBFL technique. Thus, Xie *et al.* [1] conducted a theoretical investigation of the effectiveness of risk evaluation formulas and defined two types of relations between formulas, namely, equivalent and better. They developed an innovative framework for theoretical analysis. The framework identifies the relations between different SBFL formulas by dividing all program statements into three disjoint sets to compare the sizes of these sets for different SBFL formulas. The three disjoint sets include the suspiciousness values higher than, equal to, and lower than the suspiciousness value of the faulty statement. Among the 30 investigated formulas in their study, they proved that for the single-fault scenario, there are five maximal formulas, namely, Naish1, Naish2, Wong1, Russell & Rao, and Binary. Kochhar *et al.* [35] show that 98% of practitioners consider a fault localization technique to be useful only if it reports the real faults within the top 10 of the suspicious ranking lists. However, due to the nature of SBFL, it does not always rank the root causes at the top. Thus, the researchers in [36] proposed an SBFL technique that enlarges the nonfaulty region iteratively to narrow down the suspicious region and then ranks those components in the suspicious region using existing SBFL techniques to improve the absolute ranking of faulty elements.

Fault localization techniques can be divided into two categories, including component-based and statement-based fault localization. The former is too coarse to accurately locate the real fault statement, and the latter is too complicated and costs more time. Thus, in [34], the researchers proposed a new technique for fault localization called the double-times-locating strategy. First, it built the program spectrum to abstract function traces by abstracting the function call graph from the source program, and then the function candidates were sorted by using model-based diagnosis. Second, it used DStar [11] to locate the faults in the function candidates. The approach improved the fault localization performance with respect to the average accuracy but cannot locate the accurate place when multiple statements are ranked tied. In this paper, the SMFL approach solves this problem by combining the spectrum and mutation.

Debroy and Wong [37] proposed to fix faults by combining the idea of mutation and fault localization. The statements are ranked according to their likelihood of containing faults and then are mutated in the same order to produce potential fixes. Tarantula and Ochiai are utilized for locating faults. 65 out of 314 faults are fixed by 8 selected mutation operators. Instead of fixing faults, SMFL focuses on improving the fault localization accuracy, especially in terms of $E_{inspect}@1$, to save the effort of validating the faults. Therefore, SMFL would be used to improve the efficiency of fixing faults.

B. MUTATION-BASED FAULT LOCALIZATION

MBFL is an important fault localization method, which has high accuracy, but it is complicated and costs much time. While SBFL techniques consider whether a statement is executed or not, MBFL techniques consider whether the execution of a statement affects the result of a test via a mutation test [2]. The idea of whether the execution of a statement affects the program behaviors has been researched by many studies [3], [8]–[11], [38]. Cleve and Zeller [38] first proposed to identify the state differences by comparing the program states of a failing and a passing run. They focus on the variables and values that are relevant for the failure in space and the moments of the variables begin being failures caused in time.

MBFL is based on mutants, where a mutant typically changes one statement by replacing one operand or expression with another [2], [8], [9], [34]. It calculates the mutation score of each mutant and averages the mutation scores of the mutants of a statement as the suspiciousness value of the statement. Typically, an element is suspicious if it affects failed test cases more frequently and affects passed test cases more rarely. However, MBFL costs a huge amount of time to generate and execute mutants.

Li *et al.* [10] proposed a new approach MURE for fault localization, which made use of mutation to refine spectrum-based fault localization. First, it used an SBFL technique Nash2 to output a list of suspicious statements, and then it generated mutants for candidates and estimated their likelihood of relating to faults. Second, it refined the suspicious statements list by adjusting part of the suspicious statements' ordering. The experimental result indicated that the accuracy of MURE was improved by 30% compared with that of Naish2. However, MURE only focuses on the statements that are executed by all of the failed cases. In addition, it only randomly selected at most 5 mutants for one statement candidate to estimate the mutation impact of the statement. Different from MURE, SMFL considers all the program statements. Moreover, in SMFL, the mutation score of a statement candidate is estimated based on all the mutants of the statement candidate.

Recently, MBFL has been applied to improve the effectiveness of fault localization techniques [2], [21]. Spencer *et al.* [21]. proposed an improved MBFL technique MCBFL that uses mutation coverage information and mutation kill information. It uses *EXAM* to evaluate the average rank of faulty elements. However, programmers use $E_{inspect}@n$ more to evaluate the absolute rank of the faulty elements. Zou *et al.* [2] proposed CombineFL to explore combinations of a wide range of techniques that rely on different information sources. It combines 7 types of techniques, including SBFL, MBFL, slicing, stack trace, predicate switching, information-retrieval-based, and history-based. Although it outperforms all these techniques, it costs much time to execute all the fault localization techniques and learn to rank.

V. CONCLUSION AND FUTURE WORK

This paper proposes a fault localization approach that combines the spectrum and mutation to improve the fault localization accuracy. First, the faulty program is evaluated by using SBFL, and the potential faulty statements are ranked according to their suspiciousness. Then, the mutants of the program are generated and executed by MBFL. Finally, the statements that are ranked as top tied *n* by SBFL are reranked according to their mutation scores. The proposed fault localization approach leverages the advantages of both SBFL and MBFL and improves the fault localization accuracy.

In this paper, we adopt Ochiai and DStar as the SBFL techniques and adopt MUSE and Metallaxis as the MBFL techniques. In the future, we will combine other SBFL techniques and MBFL techniques for fault localization and optimize the MBFL techniques to fit SMFL and reduce the execution time costs of mutation tests. Moreover, programs with multiple faults will be considered in future work.

ACKNOWLEDGMENT

The preliminary version of this article was presented at the 6th International Conference on Dependable Systems and Their Applications (DSA) 2019 [39].

REFERENCES

[1] 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 Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 1–40, Oct. 2013.
[2] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Trans. Softw. Eng.*, early access, Jan. 10, 2019, doi: 10.1109/TSE.2019.2892102.
[3] M. Weiser, "Program slicing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 352–357, Jul. 1984.
[4] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proc. Testing, Academic Ind. Conf. Pract. Res. Techn.*, Sep. 2007, pp. 89–98.
[5] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An empirical investigation of the relationship between spectra differences and regression faults," *Softw. Test., Verification Rel.*, vol. 10, no. 3, pp. 171–194, Sep. 2000.
[6] Y. Xiaobo, B. Liu, and W. Shihai, "An analysis on the negative effect of multiple-faults for spectrum-based fault localization," *IEEE Access*, vol. 7, pp. 2327–2347, 2019.
[7] H. He, J. Ren, G. Zhao, and H. He, "Enhancing spectrum-based fault localization using fault influence propagation," *IEEE Access*, vol. 8, pp. 18497–18513, 2020.
[8] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *Proc. IEEE 7th Int. Conf. Softw. Test., Verification Validation*, Mar. 2014, pp. 153–162.
[9] M. Papadakis and Y. Le Traon, "Metallaxis-FL: Mutation-based fault localization," *Softw. Test., Verification Rel.*, vol. 25, nos. 5–7, pp. 605–628, Aug. 2015.
[10] Z. Li, L. Yan, Y. Liu, Z. Zhang, and B. Jiang, "MURE: Making use of MU tations to RE fine spectrum-based fault localization," in *Proc. 18th Int. Conf. Softw. Qual., Rel., Secur. Companion (QRS-C)*, Aug. 2018, pp. 56–63.
[11] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The DStar method for effective software fault localization," *IEEE Trans. Rel.*, vol. 63, no. 1, pp. 290–308, Mar. 2014.
[12] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," in *Proc. 6th Int. Symp. Softw. Rel. Eng.*, 1995, pp. 143–151.
[13] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proc. 18th IEEE Int. Conf. Automated Softw. Eng.*, 2015, pp. 30–39.
[14] M. Ghorbani and M. S. Flallah, "Run-time verification for observational determinism using dynamic program slicing," in *Proc. 20th Int. Conf. Inf. Secur. (ISC)*, Nov. 2017, pp. 405–416.
[15] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, "Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, Sep. 2014, pp. 181–190.
[16] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, "CrashLocator: Locating crashing faults based on crash stacks," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, 2014, pp. 204–214.
[17] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proc. 28th Int. Conf. Softw. Eng. (ICSE)*, 2006, pp. 272–281.
[18] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 14–24.
[19] S. Kim, T. Zimmermann, E. J. Whitehead, Jr., and A. Zeller, "Predicting faults from cached history," in *Proc. 29th Int. Conf. Softw. Eng. (ICSE)*, May 2007, pp. 489–498.
[20] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu, "Bugcache for inspections: Hit or miss" in *Proc. 19th ACM SIGSOFT Symp. Found. Softw. Eng. (SIGSOFT/FSE)*, 2011, pp. 322–331.
[21] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, May 2017, pp. 609–620.
[22] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, 2011, pp. 199–209.
[23] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, Aug. 2016.
[24] R. Just, F. Schweiggert, and G. M. Kapfhammer, "MAJOR: An efficient and extensible tool for mutation analysis in a java compiler," in *Proc. 26th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2011, pp. 612–615.
[25] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proc. 20th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2005, pp. 273–282.
[26] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *Proc. 25th Int. Symp. Softw. Test. Anal. (ISSTA)*, 2016, pp. 177–188.
[27] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for java programs," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, 2014, pp. 437–440.
[28] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 15–26, Jun. 2005.
[29] E. Wong, T. Wei, Y. Qi, and L. Zhao, "A crosstab-based statistical method for effective fault localization," in *Proc. Int. Conf. Softw. Test., Verification, Validation*, Apr. 2008, pp. 42–51.
[30] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectrabased software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 1–37, Aug. 2011.
[31] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham, "Effective software fault localization using an RBF neural network," *IEEE Trans. Rel.*, vol. 61, no. 1, pp. 149–169, Mar. 2012.
[32] F. Zong, H. Huang, and Z. Ding, "Software fault location based on double-times-locating strategy," *J. Softw.*, vol. 27, no. 8, pp. 1993–2007, 2016.
[33] J. A. Jones, J. F. Bowring, and M. J. Harrold, "Debugging in parallel," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, Jul. 2007, pp. 16–26.
[34] J. S. Collofello and L. Cousins, "Towards automatic software fault location through decision-to-decision path analysis," in *Proc. Nat. Comput. Conf.*, 1986, p. 539.
[35] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proc. 25th Int. Symp. Softw. Test. Anal. (ISSTA)*, 2016, pp. 165–176.

[36] Y. Wang, Z. Huang, B. Fang, and Y. Li, ''Spectrum-based fault localization via enlarging non-fault region to improve fault absolute ranking,'' *IEEE Access*, vol. 6, pp. 8925–8933, 2018.
[37] V. Debroy and W. E. Wong, ''Combining mutation and fault localization for automated program debugging,'' *J. Syst. Softw.*, vol. 90, no. 1, pp. 45–60, Apr. 2014.
[38] H. Cleve and A. Zeller, ''Locating causes of program failures,'' in *Proc. 27th Int. Conf. Softw. Eng.*, 2005, pp. 342–351.
[39] M. Jia, Z. Cui, Y. Wu, R. Xie, and X. Liu, ''SMFL integrating spectrum and mutation for fault localization,'' in *Proc. 6th Int. Conf. Dependable Syst. Their Appl. (DSA)*, Jan. 2020, pp. 511–512.

ZHANQI CUI (Member, IEEE) received the B.E. and Ph.D. degrees in software engineering and computer software and theory from Nanjing University, in 2005 and 2011, respectively. He was a visiting Ph.D. Student with the University of Virginia from September 2009 to September 2010. He is currently an Associate Professor with Beijing Information Science and Technology University. His research interests include software analysis and testing.

MINGHUA JIA (Student Member, IEEE) received the B.E. degree in software engineer from the Computer School, Beijing Information Science and Technology University. He is currently pursuing the master's degree with the School of Information, Central University of Finance and Economics. His research interests include software engineering and software testing.

XIANG CHEN (Member, IEEE) received the Ph.D. degree in computer software and theory from Nanjing University, in 2011. He is currently an Associate Professor with the School of Information Science and Technology, Nantong University. His research interests include software defect prediction, regression testing, and combinatorial testing.

LIWEI ZHENG received the Ph.D. degree in computer software and theory from the Academy of Mathematics and Systems Science, Chinese Academy of Sciences, in 2009. He is currently an Associate Professor with Beijing Information Science and Technology University. His research interests include requirement engineering and trusted computing.

XIULEI LIU received the Ph.D. degree in computer science from the Beijing University of Posts and Telecommunications, in March 2013. He was a visiting Ph.D. Student with the CCSR, University of Surrey, from October 2008 to October 2010. He is currently an Associate Professor with Beijing Information Science and Technology University. His research interests include semantic sensor, semantic Web, knowledge graph, and semantic information retrieval.