SBFL-Suitability: A Software Characteristic for Fault Localization

Yui Sasaki*†, Yoshiki Higo*, Shinsuke Matsumoto* and Shinji Kusumoto*
*Graduate School of Information Science and Technology, Osaka University, Japan
†The Japan Research Institute, Limited, Japan
{s-yui, higo, shinsuke, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—*Spectrum-Based Fault Localization* (in short, SBFL) is one of the popular techniques to localize faulty code fragments of a given program. SBFL utilizes the information about which statements are executed by each of the success or failure test cases. There are various implementation ways for the same functionality if we use high-level programming languages. The authors consider that differences in these implementation ways may affect the efficiency of localizing faults using SBFL. In this paper, we define a characteristic to what extent a program is suitable for SBFL as *SBFL-Suitability*, and we propose a technique for measuring *SBFL-Suitability*. The proposed technique generates many slightly-variant programs from a given program with *Mutation Testing*, and then it measures how accurately SBFL detects the changed program statements in the variant programs. We conducted an experiment to investigate how *SBFL-Suitability* differs depending on the differences in source code structures. As a result, we found that (1) the fewer statements in the same nesting level, the higher *SBFL-Suitability* tends to be, and (2) the presence of *Early Return* improves *SBFL-Suitability*.**

***Index Terms*—Spectrum-Based Fault Localization, Mutation Testing, Software Quality**

I. Introduction

In software development, debugging is a highly labor-intensive and costly task [1]. Developers spend almost half or more their programming time on debbuging [2]. For this reason, there are a variety of studies on supporting debugging. *Fault Localization* is one of the promising techniques that automatically localize faulty code fragments of a given program. Recently, *Spectrum-Based Fault Localization* (in short, SBFL) has been actively studied [3]. SBFL techniques calculate the likelihood of a fault (henceforth, suspiciousness) for each statement in a given faulty program using test results and the information about which statements are executed by each test case (henceforth, execution paths). We can efficiently identify a faulty statement of a faulty program by checking the statements in the program in the descending order of their suspiciousness. The applied SBFL technique is presumed to work the most efficiently for the program if the faulty statement has the highest suspiciousness. In other words, the given program and test cases are well suited for SBFL.

Software quality consists of various viewpoints. ISO/IEC 25010 [4] defines the quality model for software products, which comprises the eight quality characteristics and sub-characteristics derived from each of them. *Maintainability*, which is one of the quality characteristics, includes *Analysability* as one of its sub-characteristics. *Analysability* indicates the degree of efficiency of diagnosing the causes of failures or identifying parts to be modified.

There are various implementation ways in high-level programming languages such as C/C++ and Java, and thus developers choose how to implement the required functionality according to their preferences or project policies. When an implementation of a functionality gets changed, the execution paths of its test cases may vary. This change would lead to differences in each statement's suspiciousness and ranking. Therefore, we believe that source code itself has a characteristic that indicates the efficiency of localizing faults.

In this paper, we define *SBFL-Suitability* as the efficiency of localizing faults using SBFL techniques. *SBFL-Suitability* can be a part of *Maintainability* or *Analyzability*. Considering *SBFL-Suitability* as a part of software quality characteristics allows us to conduct the following activities.

- We can find how reliable the SBFL results are for a given program. If it is reliable, we can debug the program with the information on SBFL.
- We can conduct refactoring to a given program from the viewpoint of improving its *SBFL-Suitability*.

We also propose a technique for measuring *SBFL-Suitability* for a given program. More concretely, firstly, the proposed technique generates many slightly-variant programs by changing a target program intentionally with *Mutation Testing* [5]. The generated programs can be considered as faulty programs. Secondly, the technique applies SBFL to each of the generated programs. Finally, the technique calculates *SBFL-Suitability* by measuring how accurately SBFL localizes the changed program statement of each of the generated programs.

In this paper, we conducted an experiment to investigate how the differences in source code structures affect *SBFL-Suitability*. As a result, we found that some source code characteristics, such as a small number of statements at the same nesting level or the presence of *Early Return*, improve *SBFL-Suitability*.

The main contributions of this paper are the following.

- We introduce a novel software quality characteristic, *SBFL-Suitability*.
- We propose a concrete technique for measuring *SBFL-Suitability* utilizing *Mutation Testing*.
- We find that *SBFL-Suitability* varies with source code structures.

- We reveal the characteristics of source code structures that improve *SBFL-Suitability*.

II. Related Work

SBFL is one of the most popular techniques in the field of fault localization and it has been actively studied recently [6]–[8]. SBFL techniques calculate suspiciousness of the program statements in a program (i.e., the probability that the statement includes a fault) using the test results and the execution paths of each test case. The execution paths are information about which statements in the source code are executed by each test case. Intuitively, a statement that is executed by many failed test cases can be considered more likely to contain a fault. Abreu et al. compared several SBFL techniques, and they concluded that the Ochiai's formula [9] was the most effective one [10].

Mutation Testing [5] is a technique to evaluate given tests using generated faulty programs, called *mutants*. A mutant is generated by applying small changes (e.g., replacement of infix operators) from the original program that behaves correctly. The key idea of the mutation testing is to measure to what extent given tests can identify the mutants as faulty programs. Note that mutants might not be faulty because the applied changes might not alter the behavior. There are various tools for mutation testing [11]–[13]. A recent study reported that mutation testing was useful for improving the verification of industry software programs [14].

III. *SBFL-Suitability*

The authors consider that source code itself has characteristics of how well it is suited to SBFL. Even if two different source code have the same functionality and test cases, their different source code structures may cause differences in the efficiency of identifying faults using SBFL. In this paper, we define to what extent a given source code is suitable for SBFL as *SBFL-Suitability*.

Herein, we show a concrete example of how *SBFL-Suitability* differs depending on source code structures. Figure 1 shows two Java methods that take two input numbers and return true if and only if at least one of them is positive. Figure 1(b) shows the method that a refactoring has been applied to the method in Figure 1(a). More concretely, the method of Figure 1(a) returns the value at the end of the method with the temporary variable `result`, which is assigned inside of each if-block, whereas the method of Figure 1(b) directly returns inside of each if-block. Both of the methods contain the same fault. Test case t_4 fails because the conditional expression of s_4 in Figure 1(a) and the one of s'_4 in Figure 1(b) behave incorrectly when the variable `b` is 0.

Figures 1(a) and 1(b) show test results (P: Passed, F: Failed), the execution path for each test case, and the suspiciousness[1] (labeled as 'susp') of each program statement calculated by the Ochiai's formula. Suspiciousness takes a value between 0 and 1. Value 1 means the highest likelihood that the fault locates at its statement. Those results show that faulty statements s_4 and s'_4 have different suspiciousness.

[1]The prepared test suite does not execute s'_6 in Figure 1(b), which means that the suspiciousness is not calculated for s'_6.

	program	*susp*	test case (input: a, b)			
			t_1 (1,1)	t_2 (1,0)	t_3 (0,1)	t_4 (0,0)
(a) Before Refactoring	s_1: **boolean** result = **false**;	0.50	✓	✓	✓	✓
	s_2: **if** (0 < a)	0.50	✓	✓	✓	✓
	s_3: result = **true**;	0.00	✓	✓		
	🐞 s_4: **if** (0 <= b) // correct: 0 < b	0.50	✓	✓	✓	✓
	s_5: result = **true**;	0.50	✓	✓	✓	✓
	s_6: **return** result;	0.50	✓	✓	✓	✓
(b) After Refactoring	s'_2: **if** (0 < a)	0.50	✓	✓	✓	✓
	s'_3: **return true**;	0.00	✓	✓		
	🐞 s'_4: **if** (0 <= b) // correct: 0 < b	0.71			✓	✓
	s'_5: **return true**;	0.71			✓	✓
	s'_6: **return false**;	-				
	test results:		P	P	P	F

Fig. 1. SBFL results compared with different source code structures

We consider a situation that a developer who does not know the faulty locations in those methods tries to identify the faulty statements on the basis of the SBFL results. Then, s/he checks each of the statements in the descending order of their suspiciousness. In the case of Figure 1(a), the suspiciousness of faulty statement s_4 is the highest though there are a total of five statements with the same highest suspiciousness. Thus, in the worst case, s/he has to check the five statements to identify the fault. In contrast, in Figure 1(b), s/he can identify the fault by only checking at most two statements. Therefore, the method of Figure 1(b) is more efficient in identifying the fault than the method of Figure 1(a). Note that this scenario is not depending on the SBFL formula (i.e., Ochiai) because their execution paths itself are changed by refactoring.

IV. Measuring *SBFL-Suitability*

We propose a technique to measure *SBFL-Suitability* of a given program. Our basic idea is to generate multiple faulty programs from the original program and measure how efficiently SBFL can identify which statement is faulty. To generate faulty programs, we utilize mutation testing techniques. The proposed technique measures how accurately SBFL can localize the faulty statements in each of the generated mutants while mutation testing essentially measures how accurately a test suite can identify the generated mutants as faulty.

A. *Factors Affecting SBFL-Suitability*

SBFL-Suitability depends on not only the source code structure of a program but also the following two factors: a test suite and a mutant generator.

As an example, we consider the situation where the test suite includes only t_3 and t_4 in Figure 1(b). Since the execution paths for the two test cases are identical, the suspiciousness calculated from each of the execution paths are the same for at least statements s'_2, s'_4, and s'_5 that are executed by those test cases. In the case of the original test suite, statements s'_4 and s'_5 are more likely to be fault than s'_2. In this way, how efficiently SBFL works varies depending on the given test suite.

The proposed technique uses several *mutation operators*, which are transformation rules for generating mutants. This

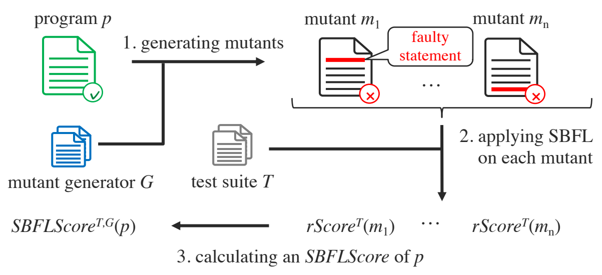

Fig. 2. Process of the *SBFLScore* Calculation

paper calls a set of mutation operators as a *mutant generator*. *SBFL-Suitability* depends on a mutant generator.

B. Calculating SBFLScore

We define an indicator of the degree of *SBFL-Suitability* as *SBFLScore*. *SBFLScore* takes a value between 0 and 1, and the closer to 1, the higher *SBFL-Suitability*. Let $SBFLScore^{T,G}(p)$ be an *SBFLScore* of a program p with a test suite T and a mutant generator G. Figure 2 shows the process of the *SBFLScore* calculation. We calculate an *SBFLScore* of p with the following three steps:

1) generating multiple mutants from p,
2) applying SBFL to each mutant and then calculating suspiciousness ranking of their faulty statements, and
3) calculating an *SBFLScore* of p from the suspiciousness ranking of each of the generated mutants.

Step 1. Generating Mutants: The proposed technique generates multiple mutants using a mutant generator G for a program p. Each mutant has only a single different statement from the original program. We treat such a statement as a faulty statement. Note that, at this point, we do not know whether all of the mutants are faulty or not.

Step 2. Applying SBFL for Each Mutant: For each of the generated mutants, we apply SBFL using a test suite T, and then, we calculate a suspiciousness for each statement. We exclude mutants that are identified as non-faulty. Let $M^G(p)$ be a set of obtained mutants for a program p with a mutant generator G. We define the following values for statement s included in mutant $m \in M^G(p)$:

- $susp^T(s)$: the suspiciousness of s when executing T,
- $rank^T(s)$: the rank of $susp^T(s)$, and
- $rScore^T(s)$: the normalized $rank^T(s)$ between 0 and 1.

We use Ochiai's formula, known to be effective, to calculate suspiciousness. $susp^T(s)$ is calculated by the Ochiai's formula as follows:

$$susp^T(s) = \frac{fail^T(s)}{\sqrt{totalFail^T * (fail^T(s) + pass^T(s))}}$$

where $fail^T(s)$ is the number of failed test cases that cover s, $pass^T(s)$ is the number of passed test cases that cover s, and $totalFail^T$ is the total number of failed test cases.

Next, we calculate $rank^T(s)$, which indicates the number of statements whose suspiciousness is equal to or greater than $susp^T(s)$. For example, if there are two statements with suspiciousness 1.0 and one statement with 0.9, both statements with 1.0 are in the second place, and the statement with 0.9 is in the third place.

A rank of suspiciousness has different worthiness depending on the total number of statements. For example, the 10th place out of 100 statements is more valuable than the 10th place out of 10 statements. Thus, we normalize a rank of suspiciousness to a range between 0 and 1. A normalized rank $rScore^T(s)$ indicates how high it appears in the total number of statements and is calculated as follows:

$$rScore^T(s) = 1 - \frac{rank^T(s) - 1}{totalStatements^T - 1}$$

where $totalStatements^T$ is the number of statements executed by a test suite T. The value 1 is the most valuable.

Let $rScore^T(m)$ be the normalized rank of the faulty statement in mutant m. $rScore^T(m)$ is unique for each mutant because each mutant contains only a single faulty statement. Let $s^m{}_{fault}$ be the faulty statement in a mutant m. $rScore^T(m)$ is the unique normalized rank of statement $s^m{}_{fault}$ as follows:

$$rScore^T(m) = rScore^T(s^m{}_{fault})$$

Step 3. Calculating SBFLScore: SBFLScore is the average value of the normalized rank of each mutant's faulty statement generated from p. We define it by the following formula.

$$SBFLScore^{T,G}(p) = \frac{1}{|M^G(p)|} \sum_{m \in M^G(p)} rScore^T(m)$$

V. Experiment

We implemented a tool based on the proposed technique for Java. We conducted an experiment to investigate how the differences in source code structures affect *SBFL-Suitability*.

A. Experimental Settings

We used PIT [15], an open-source mutation testing tool, as a reference for mutation operators. PIT has published the transformation rules of mutation operators, and it defines groups of operators on its web site. We selected all of the eleven mutation operators included in group 'DEFAULTS'. We implemented a new mutant generator tool for source code that supports these operators because PIT (and other existing mutation testing tools) generate bytecode mutants instead of source code ones. Table I shows the target mutation operators.

In this paper, we focus on refactoring for a difference in source code structures. Table II shows the five types of refactoring. We selected[2] refactorings classified as '*Simplifying Conditional Expression*' [16]. The reason is that the execution path of each test case is changed because statements in conditional blocks are changed by these refactorings, which might affect their suspiciousness. We manually and thoughtfully created target programs by reference to program sample of each refactoring pattern [16] as many mutation operators

[2]We excluded refactorings across multiple Java methods because we apply SBFL and then calculated a ranking of the statements for each method.

could be applied as possible. We also manually created test cases with satisfying the condition coverage for all the mutants generated from each program.

B. Results and Discussion

We applied our tool to the prepared programs. Note that in all target programs, the generated mutants failed at least one or more test cases; in other words, they were faulty. Table II shows the results of the *SBFLScore* measurement. *SBFLScore* was decreased by the refactoring in Cases 1–3 whereas *SBFLScore* was increased in Cases 4 and 5. Due to space limitations, we discuss only Cases 1 and 5.

Figures 3 and 4 shows the programs of Cases 1 and 5. In both figures, (a) and (b) show the code fragments before and after refactoring, respectively. We describe a pair of statements with the same subscript number before and after refactoring such s_2 and s'_2 in Figure 3 as $\langle s_2, s'_2\rangle$. Such a pair denotes that they have a correspondence relation between before and after refactoring. For example in Figure 3, s'_2 has not been changed from s_2 through the refactoring whereas s'_{1a} and s'_{1b} have been split from s_1 through the refactoring. These relations are described as $\langle s_2, s'_2\rangle$, $\langle s_1, s'_{1a}\rangle$, and $\langle s_1, s'_{1b}\rangle$. Figures 3 and 4 also show *rScore* of each statement included in each of the generated mutants. Each *rScore* is also represented by a horizontal bar chart whose range is from 0 to 1. *rScore* of the faulty statement in each mutant is shown in bold. For example, m_1 in Figure 3(a) is the mutant where mutation operator NC has been applied to statement s_1, and its *rScore* is 0.67.

In the following description, we focus on a pair of mutants with the same subscript number before and after refactoring, such as m_1 and m'_1. We describe such a pair as $\langle m_1, m'_1\rangle$. Such a pair of mutants denotes that they have been generated by applying the same mutation operators to corresponding statements between before and after the refactoring (e.g., $\langle s_1, s'_{1a}\rangle$). By comparing *rScore* of each pair of mutants, we investigate how different the ease of localizing the corresponding faulty statement between before and after the refactoring.

1) Case 1: '*Decompose Conditional*'

'*Decompose Conditional*' is a refactoring that extracts conditional expressions to meaningfully named methods. We prepared the target program whose conditional expressions were extracted to variables instead of methods because our technique measures *SBFL-Suitability* for each method.

TABLE I
MUTATION OPERATORS

Mutation operator		Transformation example	
		Before	After
(CB)	Conditional Boundary	`a < b`	`a <= b`
(INC)	Increments	`n++`	`n--`
(INV)	Invert Negatives	`-n`	`n`
(MA)	Math	`a + b`	`a - b`
(NC)	Negate Conditionals	`a < b`	`a >= b`
(VM)	Void Method Calls	`method();`	`;`
(PR)	Primitive Returns	`return 5;`	`return 0;`
(ER)	Empty Returns	`return "str";`	`return "";`
(FR)	False Returns	`return true;`	`return false;`
(TR)	True Returns	`return false;`	`return true;`
(NR)	Null Returns	`return object;`	`return null;`

	rScore						
mutant:	m_1	m_2	m_3	m_4	m_5	m_6	m_7
mutation operator:	NC	CB	INC	NC	CB	INC	PR
s_1: **if** (0 < n)	**0.67**	**0.50**	0.50	0.50	0.25	0.25	0.75
s_2: n--;	0.33	1.00	**1.00**	0.00	0.00	0.00	0.25
s_3: **else if** (n < 0)	0.00	0.00	0.00	**1.00**	**0.75**	0.75	0.00
s_4: n++;	-	0.00	0.00	0.25	0.75	**1.00**	0.25
s_5: **return** n;	0.67	0.50	0.50	0.50	0.25	0.25	**0.75**

(a) Before Refactoring (*SBFLScore* =0.81)

	rScore						
mutant:	m'_1	m'_2	m'_3	m'_4	m'_5	m'_6	m'_7
mutation operator:	NC	CB	INC	NC	CB	INC	PR
s'_{1a} : **boolean** f1 = (0 < n);	**0.40**	**0.33**	0.33	0.33	0.17	0.17	0.50
s'_{3a} : **boolean** f2 = (n < 0);	0.40	0.33	0.33	**0.33**	**0.17**	0.17	0.50
s'_{1b} : **if** (f1)	0.40	0.33	0.33	0.33	0.17	0.17	0.50
s'_2 : n--;	0.20	1.00	**1.00**	0.00	0.00	0.00	0.17
s'_{3b} : **else if** (f2)	0.00	0.00	0.00	1.00	0.83	0.83	0.00
s'_4 : n++;	-	0.00	0.00	0.17	0.83	**1.00**	0.17
s'_5 : **return** n;	0.40	0.33	0.33	0.33	0.17	0.17	**0.50**

(b) After Refactoring (*SBFLScore* =0.53)

Fig. 3. Case 1: Decompose Conditional

SBFLScore was decreased by the refactoring. *rScore* of the faulty statements were the same for $\langle m_3, m'_3\rangle$ and $\langle m_6, m'_6\rangle$ whereas they were decreased for the others. For example, *rScore* of the faulty statements in $\langle m_1, m'_1\rangle$ were the highest in each mutant. There were two statements with the same highest *rScore* in m_1 (i.e., s_1 and s_5) whereas there were four statements in m'_1 (i.e., s'_{1a}, s'_{3a}, s'_{1b} and s'_5). Those statements are at the nesting level 1. We checked the calculation process of *rScore*, and then we found that *susp* of all of the statements were 1.00. This observation implies that the increase in the number of the same nesting level leads to the decrease of *rScore*; as a result, *SBFL-Suitability* gets worsened.

The fewer statements at the same nesting level, the higher *SBFL-Suitability* tends to be.

2) Case 5: '*Replace Nested Conditional with Guard Clauses*'

'*Replace Nested Conditional with Guard Clauses*' is a refactoring that returns early from a method by checking conditions that are satisfied not to execute the main process of the method. The refactoring is effective to prevent the source code from being deeply nested. To simplify the experiment, we prepared the target program where a return statement has been inserted in each conditional block. Such a coding style is called '*Early Return*' [17].

SBFLScore was increased by the refactoring. *rScore* of

TABLE II
TARGET REFACTORINGS

Case	Refactoring pattern	*SBFLScore*	
		Before	After
Case 1	Decompose Conditional	0.81	0.53
Case 2	Consolidate Conditional Expression	0.95	0.72
Case 3	Consolidate Duplicate Conditional Fragments	0.69	0.53
Case 4	Remove Control Flag	0.61	0.67
Case 5	Replace Nested Conditional with Guard Clauses	0.83	0.95

	rScore							
mutant:	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8
mutation operator:	NC	CB	INV	NC	CB	INV	INV	PR
s_1: **int** result = 0;	0.67	0.50	0.50	0.33	0.33	0.33	0.33	0.67
s_2: **if** (x > 0)	**0.67**	**0.50**	0.50	0.33	0.33	0.33	0.33	0.67
s_3: result = -10;	0.50	1.00	**1.00**	0.00	0.00	0.00	0.00	0.17
s_4: **else if** (y > 0)	0.33	0.00	0.00	**1.00**	**0.83**	0.83	0.83	0.50
s_5: result = -20;	0.00	0.00	0.00	0.83	1.00	**1.00**	0.00	0.00
else	-	-	-	-	-	-	-	-
s_6: result = -30;	0.17	0.00	0.00	0.17	0.00	0.00	**1.00**	0.33
s_7: **return** result;	0.67	0.50	0.50	0.33	0.33	0.33	0.33	**0.67**

(a) Before Refactoring (*SBFLScore* =0.83)

	rScore									
mutant:	m'_1	m'_2	m'_3	m'_4	m'_5	m'_6	m'_7	m'_8	m'_9	m'_{10}
mutation operator:	NC	CB	INV	NC	CB	INV	INV	PR	PR	PR
s'_2: **if** (x > 0)	**1.00**	**0.75**	0.75	0.50	0.50	0.50	0.50	0.50	0.75	0.50
s'_3: **return** -10;	0.75	1.00	**1.00**	0.00	0.00	0.00	0.00	0.00	**1.00**	0.00
s'_4: **if** (y > 0)	0.50	0.00	0.00	**1.00**	**0.75**	0.75	0.75	0.75	0.00	0.75
s'_5: **return** -20;	0.00	0.00	0.00	0.75	1.00	**1.00**	0.00	0.00	0.00	**1.00**
s'_6: **return** -30;	0.25	0.00	0.00	0.25	0.00	0.00	**1.00**	**1.00**	0.00	0.00

(b) After Refactoring (*SBFLScore* =0.95)

Fig. 4. Case 5: Replace Nested Conditional with Guard Clauses

the faulty statements were the same or increased after the refactoring except for $\langle m_5, m'_5 \rangle$. s_1, s_2, and s_7 were at the nesting level 1 and these *rScore* were always the same in each mutant. In contrast, s'_2, s'_4, and s'_6 were also at the nesting level 1 while these *rScore* were different from each other. The reason was that the number of test cases that executed those statements was different due to the inserted return statements. This observation implies that applying *Early Return* reduces the number of statements with the same suspiciousness; as a result, *SBFL-Suitability* is improved.

The presence of *Early Return* improves *SBFL-Suitability*.

VI. Threats to Validity

Our experiment selected the five types of refactorings listed in Table II. The target programs were simple, which were manually created as examples of the refactoring patterns. If we conduct experiments with other types of refactorings or real programs, we may obtain different results and discover new characteristics of source code structures.

SBFLScore depends on a test suite and a mutant generator, as mentioned in Section IV. We selected the eleven mutation operators listed in the Table I as the mutant generator. If we use more mutation operators, we may obtain different results. We also created manually test suites that satisfied the condition coverage of each of the programs and their mutants. There are test cases that take the same execution path when executing the test for each mutant. It is possible that such test cases affect the results of our experiment.

VII. Conclusion

In this paper, we considered that source code has characteristics of how well it is suited to SBFL, and then we defined the characteristic as *SBFL-Suitability*. Besides, we proposed a technique to measure *SBFL-Suitability* utilizing mutation testing techniques. We applied the proposed technique to several refactoring situations, and observed the difference of *SBFL-Suitability* between before and after the refactoring. As a result, we found the characteristics of the source code structures that improve *SBFL-Suitability*.

In the future, we are going to conduct experiments for real and large-scale programs to generalize our proposed technique. Besides, we are going to propose a technique to convert source code structures to improve *SBFL-Suitability* of a given program. If we convert source code structures to a higher degree of *SBFL-Suitability* before applying SBFL, we can identify faulty location more efficiently.

Acknowledgment

This study has been supported by Grants-in-Aid for Scientific Research (B) (18H03222), Scientific Research (B) (20H04166) from the Japan Society for the Promotion of Science.

References

[1] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification," *IBM Systems Journal*, vol. 41, no. 1, pp. 4–12, 2002.

[2] T. Britton, L. Jeng, G. Carver, and P. Cheak, "Reversible debugging software "quantify the time and cost saved using reversible debuggers"," 2013.

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

[4] *ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*, Std., 2011.

[5] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE TSE*, vol. 37, no. 5, pp. 649–678, 2011.

[6] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proc. ICSE*, 2002, pp. 467–477.

[7] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Proc. DSN*, 2002, pp. 595–604.

[8] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for Java," in *Proc. ECOOP*, 2005, pp. 528–550.

[9] A. da Silva Meyer, A. A. F. Garcia, A. P. de Souza, and C. L. de Souza Jr., "Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (zea mays l)," *Genetics and Molecular Biology*, vol. 27, no. 1, pp. 83–91, 2004.

[10] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proc. TAIC PART*, 2007, pp. 89–98.

[11] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "MuJava: A mutation system for Java," in *Proc. ICSE*, 2006, pp. 827–830.

[12] R. Just, "The major mutation framework: Efficient and scalable mutation analysis for Java," in *Proc. ISSTA*, 2014, pp. 433–436.

[13] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: A practical mutation testing tool for Java (demo)," in *Proc. ISSTA*, 2016, pp. 449–452.

[14] R. Ramler, T. Wetzlmaier, and C. Klammer, "An empirical study on the application of mutation testing for a safety-critical industrial software system," in *Proc. SAC*, 2017, pp. 1401–1408.

[15] H. Coles. PIT. [Online]. Available: https://pitest.org/

[16] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[17] D. Boswell and T. Foucher, *The Art of Readable Code: Simple and Practical Techniques for Writing Better Code*. O'Reilly Media, 2011.