Revisiting Program Suitability for Fault Localization with Large Dataset and Various Mutation Operators

Hikaru Kubo*, Yoshiki Higo* and Shinji Kusumoto*
*Graduate School of Information Science and Technology, Osaka University, Japan
{ hkr-kubo@ist., higo@ist., kusumoto.shinji.ist@ } osaka-u.ac.jp

***Abstract*—Spectrum-Based Fault Localization (in short, SBFL) is one of the popular techniques to localize faulty statements of a given program. SBFL utilizes the information about which statements are executed in each of the successful or failed test cases. Even if multiple programs have the same functionality, the accuracy of SBFL can differ due to their structural variations. Thus, changing program structures to be suitable for SBFL may improve the accuracy of fault localization while maintaining functionality. In previous research by Sasaki et al., *SBFL-Score* was proposed to discover program structures suitable for SBFL. *SBFL-Score* is one of the metrics used to evaluate how well a program is suitable for SBFL. Furthermore, the previous research measured *SBFL-Score*s for pairs of programs with different structures but the same functionality, and obtained a program structure suitable for SBFL. However, a small number of programs and a small number of mutation operators used in the experiments were shortcomings. Thus, in this study, we conducted an experiment with approximately 36 times more programs and about 2.5 times more mutation operators than in the previous research. As a result of this experiment, we identified four new program structures suitable for SBFL.**

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

I. Introduction

In software development, debugging is a highly labor-intensive and costly task. There is a report suggesting that debugging accounts for over half of the costs involved in software development [1], [2]. For this reason, there are many studies supporting debugging. One area of research in debugging support is Fault Localization, which aims to localize faulty statements in a program. In recent years, Spectrum-Based Fault Localization (SBFL) has been actively studied [3]. SBFL techniques calculate the likelihood of a fault (henceforth, suspiciousness) for each program statement in a given faulty program using test results and the information about which program statements are executed in each test case (henceforth, execution paths). The basic idea behind SBFL is that statements executed in many failed tests are more likely to be faulty, while statements executed in many successful tests are less likely to be faulty.

The accuracy of SBFL is influenced by various factors, such as types of defects [4] and types of tests [5]. In these factors, the previous research by Sasaki et al. [6] focused on program structure.

The previous research proposed *SBFL-Suitability* as a quality indicator of how well a given program is suitable for SBFL [6]. Furthermore, the previous research also proposed *SBFL-Score* as an evaluation metric for *SBFL-Suitability*. The basic idea behind measuring *SBFL-Score* is to create artificial defects for a program that passes all test cases intentionally. By measuring how accurately SBFL can identify these artificial defects, the *SBFL-Suitability* of the target program can be evaluated.

The previous research suggested that the accuracy of SBFL is affected by the program structures, and it is possible to improve the accuracy of SBFL by changing the program structure temporarily before executing SBFL. Building on this concept, the previous study conducted an experiment to identify program structures that are suitable for SBFL by measuring *SBFL-Score*s using five pairs of programs with different structures but the same functionality. However, the number of programs used for measurement was only ten, and the program structures that are suitable for SBFL were not sufficiently investigated. Additionally, only 11 types of mutation operators were used to intentionally create artificial defects in programs. Due to the small-size experiment, we considered that the measured *SBFL-Score*s are inappropriate as an evaluation metric of *SBFL-Suitability*.

To address these shortcomings, we utilize a dataset [7] consisting of a large number of program pairs with different structures but the same functionality. In this study, we aim to discover new program structures that are suitable for SBFL by measuring *SBFL-Score*s using 365 programs in the dataset. Additionally, we define 16 new mutation operators to increase the number of mutants. This allows 27 mutation operators applicable to the programs.

As a result of adding new mutation operators, the number of mutants increased in approximately 98.6% of the programs. Moreover, by measuring the *SBFL-Score*s of the 365 programs and visually checking the results, we identified four new program structures that are suitable for SBFL.

II. BackGround

A. Spectrum-Based Fault Localization (SBFL)

Fault Localization is one of the techniques used to support debugging, which aims to localize faulty statements in a program. Spectrum-Based Fault Localization (henceforth, SBFL) is one of the automated fault localization techniques that utilize tests. In SBFL, the spectrum is the execution path information that indicates which statements were executed by each test case. The basic idea behind SBFL is that statements executed by many failed tests are more likely to be faulty, while statements executed by many successful tests are less likely to be faulty.

We explain the process of identifying faulty statements using SBFL. First, we execute all test cases and the success or failure of each test case and the execution path information is recorded. Next, using this information, suspiciousness is calculated for each statement. There are various techniques to calculate suspiciousness. Abreu et al. evaluated the effectiveness of calculation formulas used in SBFL and concluded that Ochiai's [8] is the most effective [9].

Formula (1) shows how suspiciousness $susp(s)$ is calculated in Ochiai. Here, $fail(s)$ represents the number of failed tests executing statement s, $pass(s)$ represents the number of successful tests executing statement s, and $totalFail$ represents the total number of failed tests.

$$susp(s) = \frac{fail(s)}{\sqrt{totalFail \times (fail(s) + pass(s))}} \tag{1}$$

The suspiciousness is calculated for all statements s; the higher the value, the more likely it to be faulty.

In calculating suspiciousness, the execution path information of failed tests is the most important factor. That is because which statements were executed and which were not in failed tests are major clues to localizing the faulty statement. This is why the numerator of Ochiai's formula is $fail(s)$, emphasizing the importance of the execution path information of failed tests.

B. SBFL-Suitability

SBFL-Suitability was proposed in the previous research [6]. *SBFL-Suitability* is one of the quality characteristics of a program, indicating how well a given program is suitable for SBFL. Even if two programs have the same functionality and test suite, differences in program structure can lead to variations in the accuracy of fault localization using SBFL.

We explain how changes in *SBFL-Suitability* occur due to differences in program structure with an example. Programs (a) and (b) shown in Figure 1 have the same functionality but different structures. When SBFL is applied to both programs using the test (c) shown in Figure 1, suspiciousness is calculated for each statement.

In program (a), there are four statements with the same suspiciousness as the faulty statement. On the other hand, in program (b), there is only one statement with the same suspiciousness as the faulty statement. The fewer statements with the same suspiciousness as the location of the defect, the fewer statements need to be checked. This means higher accuracy in fault localization using SBFL. Therefore, program (b) has higher *SBFL-Suitability* than program (a).

	program	*susp*	test case (input :a,b)			
			t_1(1,1)	t_2(1,0)	t_3(0,1)	t_4(0,0)
(a) Befor e 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 program structures

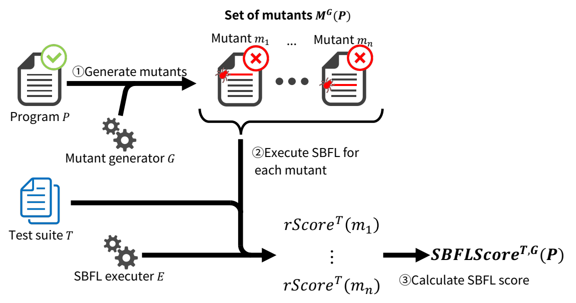


Fig. 2: Calculation process of the *SBFL-Score*

C. SBFL-Score

In addition to *SBFL-Suitability*, we explain the *SBFL-Score* proposed in the previous research [6]. The *SBFL-Score* is one of the evaluation metrics for *SBFL-Suitability*. Figure 2 shows the calculation process of *SBFL-Score*. The basic idea behind measuring the *SBFL-Score* is to utilize mutation testing techniques [10] to intentionally change a given program and create artificial defects as many as possible. Intentionally changed programs are called mutants. By measuring how accurately SBFL can identify these artificial defects, we can assess how well the entire program is suitable for SBFL.

D. Experiment in the Previous Research

The previous research conducted an experiment to investigate how *SBFL-Scores* vary due to the differences in program structures [6].

1) Overview of Experiment: The previous research conducted an experiment to measure SBFL-Scores for five program pairs by using an SBFL-Score measurement tool. Two programs in each program pair have the same functionality but different program structures. They compared the *SBFL-Scores* of the two programs in each pair and assumed that a program having a higher SBFL-Score is more suitable for SBFL than the other program in each pair. By inspecting the executing path of each program, they considered the reason why SBFL-Scores differ from the difference in program structure.

2) Experimental Targets and Test Suites: In the previous research, the experimental target was five method pairs and test suites. The five method pairs were created based on refactoring patterns by Sasaki et al. The refactoring patterns selected for this experiment were from the category of *'Simplifying Conditional Expression'* as classified by Fowler [11].

The test suite for each method was created by Sasaki et al. to satisfy the following three criteria:

- each test suite in a method pair has the same test cases,
- each mutant fails in any test case, and
- condition coverage reaches 100 %.

3) SBFL-Score measurement tool: In the previous research by Sasaki et al., they developed an SBFL-Score measurement tool. It consists of a mutant generator and an SBFL executer.

They implemented 11 types of mutation operators listed in Table I in the mutant generator. These operators were selected from the default mutation operators provided by PIT [12], an open-source mutation testing tool. PIT is widely used in mutation testing for generating mutants [13].

Their tool has a functionality to execute SBFL for mutants generated by the mutant generator. Their tool uses Ochiai's formula (Formula (1)) to calculate the suspiciousness value of each line in a mutant. Their tool also has a functionality to save the logs of SBFL such as execution path information of each mutant and SBFL-Score.

4) Result: The previous research revealed that the fewer statements exist at the same nesting level, the higher *SBFL-Suitability* tends to be.

They also got two transformation ways of program structure to improve *SBFL-Suitability* below:

- Using return statement to terminate method early.
- Reduce the number of statements in the same nesting level.

III. Experimental Setup

A. Chenges from previous research

In this research, we conduct an experiment similar to the previous research by Sasaki et al. [6].

To reveal new program structures that are suitable for SBFL, we made some changes to the experimental setup of previous research.

1) Definition of Program Structure: In the previous research by Sasaki et al., they did not define the precise meaning of program structure. In this research, we define the program structure of the method as the execution paths for the same test suite. Even if methods are written differently, if the execution paths of the two methods are the same, the program structures of the two methods are regarded as the same.

TABLE I: Mutation operators used in the previous research

Mutation Operator	Previous	Later
Conditional Boundary	`a<b`	`a<=b`
Increments	`n++`	`n--`
Invert Negatives	`-n`	`n`
Math	`a+b`	`a-b`
Negate Conditionals	`a==b`	`a!=b`
Void Method Calls	`method();`	`;`
Primitive Returns	`return 5;`	`return 0;`
Empty Returns	`return "str";`	`return "";`
False Returns	`return true;`	`return false;`
True Returns	`return false;`	`return true;`
Null Returns	`return object;`	`return null;`

	Program (Input: a)	t_1	t_2	t_3
s_1	`if (a == 0)`	✓	✓	✓
s_2	`  return 0;`			✓
s_3	`else if (a > 0)`	✓	✓	
s_4	`  return 1;`	✓		
s_5	`return -1;`		✓	✓

(a)

	Program (Input: a)	t_1	t_2	t_3
s'_1	`if (a > 0)`	✓	✓	✓
s'_2	`  return 1;`	✓		
s'_3	`else if (a < 0)`		✓	✓
s'_4	`  return -1;`		✓	
s'_5	`return 0;`			✓

(b)

Fig. 3: A method pair with different execution paths

	Program (Input: a)	t_1	t_2	t_3
s_1	`if (a>0)`	✓	✓	✓
s_2	`  return 1;`	✓		
s_3	`if (b<0&&a!=0)`		✓	✓
s_4	`  return -1;`		✓	
s_5	`return 0;`			✓

(a)

	Program (Input: a)	t_1	t_2	t_3
s'_1	`if (a>0)`	✓	✓	✓
s'_2	`  return 1;`	✓		
s'_3	`else if (a<0&&b<0)`		✓	✓
s'_4	`  return -1;`		✓	
	`else`			
s'_5	`  return 0;`			✓

(b)

Fig. 4: A method pair with same execution paths

The term "same execution paths" refers to two methods satisfying the following three conditions, where s_i represents the i-th statement in the first method, and s'_i represents the i-th statement in the second method.

- The number of all statements in the two methods is the same.
- The number of statements executed in the same test case (m) is the same in the two methods.
- For all test cases in the test suite, for range in $1 \leq i \leq m$, when statement s_i is executed, s'_i is also executed.

Figure 3 shows an example of a method pair with different execution paths, while Figure 4 shows an example of a method pair with the same execution paths. For the method pairs with the same execution paths, applied mutation operators are only the factors that affect *SBFL-Scores*. We considered such method pairs inappropriate for the experiment in this research.

2) Experimental target and test suite: The previous research used a dataset consisting of only ten methods that were created manually, which were part of five method pairs. In short, the size of the dataset is small. Thus, the generalizability of the research result is low. The previous research also has not thoroughly investigated program structures contributing to higher *SBFL-Scores*.

To address this shortcoming, we utilize a dataset [7] that includes methods with the same functionality implemented in Java. The dataset consists of 728 methods and test suites. Each method in this dataset was collected from open-source software. The methods in the dataset are categorized into 276 groups based on their functionality, with each group containing two to twelve methods. The test suite for each method was automatically generated using Evosuite [14].

We set four conditions for the experimental target.

Cond. 1: The methods within the same group share the same functionality.

Cond. 2: The test suite attached to the method reaches 100% of the instruction and condition coverages for the methods.

Cond. 3: The test suite attached to the method does not contain Flaky Test[15].

Cond. 4: The program structures in the same group are different from one another.

The Reasons for setting Cond. 2 to 4 are as follows:

Regarding Cond. 2: Ensuring 100% instruction coverage is necessary to calculate suspiciousness values from all the statements. Ensuring 100% condition coverage is necessary to attach different suspiciousness to different program statements as much as possible.

Regarding Cond. 3: A Flaky Test is a test case that exhibits both a passing and a failing result with the same code. If Flaky Tests exist in a test suite, *SBFL-Score* of a given method may vary every time we execute the test suite.

Regarding Cond. 4: The purpose of this research is to reveal what kind of program structures are more suitable for SBFL, which means that it is not necessary to treat multiple methods that have the same structure.

The procedure for obtaining methods satisfying all the conditions is as follows:

Step 1: Add some test cases manually.

The test suite for each method was automatically generated using Evosuite [14]. However, there are some cases where the automatically generated test suites do not achieve 100% instruction and condition coverage. Therefore, we manually add the test cases to ensure 100% instruction coverage and condition coverage.

Step 2: Exclude methods that do not meet the conditions.

We execute the test suite to investigate whether each target method satisfies Cond. 1 to Cond. 4. As a result, 363 methods within 145 groups were excluded from the experimental target. We use 365 methods within 131 groups that satisfy the four conditions as the experimental target.

3) Mutant generator: The previous research used 11 types of mutation operators. They implemented the ten target programs so that these mutation operators could be applied in many parts of the code, resulting in a sufficient number of mutants generated from a method. However, it may not be possible to generate a sufficient number of mutants when using other programs. In such cases, *SBFL-Score* values are less reliable.

To address this shortcoming, we visually inspected the experimental target and extracted locations where existing mutation operators cannot be applied. We defined additional mutation operators shown in Table II to mutate to the locations we described above. The reasons for selecting those mutation operators are below.

- Applicable to multiple methods.
- Not necessary to record identifiers to implement a mutation operator.

Newly defined mutation operators may generate the same mutants as existing ones. For example, if there is a statement `return false;` in a program, both the `Change Boolean Literal` and `True Return` operators can generate the same mutant. Thus, we modified the mutant generator to ensure the same mutant is not duplicately generated.

To evaluate that the *SBFL-Score* values become more reliable, we measure the number of mutants generated by additional mutation operators for each method in the experimental target. It is necessary to consider the size of the method because as the size of the method increases, the number of mutants tends to increase. To evaluate the increase in mutants compared to the method size, we calculate the *AMPL* (the number of Additional Mutants Per LOC) metric for each method. Equation (2) is the definition of *AMPL*. The term "Logical LOC" in Equation (2) refers to the number of lines in the method excluding empty lines, comment lines, and lines containing only brackets. A higher value of *AMPL* indicates improved accuracy of *SBFL-Score*s.

$$AMPL = \frac{\text{The number of additional mutants}}{\text{Logical LOC of a method}} \tag{2}$$

We implemented the 11 mutation operators mentioned in Section II-D and the additional 16 mutation operators mentioned in Section III-A3 within the mutant generator. To measure the *AMPL* value of each target program, we also implemented the functionality to switch mutation operators used. It allows us to record the difference in the number of mutants before and after adding new operators.

B. The procedure of experiment

We evaluate the increase in mutants and find factors contributing to higher *SBFL-Score*s in the following steps:

Step A: Measure *SBFL-Score*s and *AMPL*s of the target methods.

For all the target methods, we measure the *SBFL-Score*s and *AMPL*s using a measurement tool that we developed. To obtain the execution path information necessary for calculating *SBFL-Score*s, we utilized kGenProg [16], an APR tool. As mentioned in II-D3, the execution path information of running SBFL for each mutant is saved to files. It allows us to analyze the factors contributing to higher *SBFL-Score*s in Step C.

TABLE II: Additional mutation operators we implemented. If code exists in the "Later" column separated by commas, it means the mutation operator in this row generates multiple mutants.

Mutation Operator	Previous	Later	Times
Change String Literal	`"String"`	`"String1"`	96
Change Instanceof	`a instanceof A`	`a instanceof B`	192
Nonvoid Method Calls	`a=method()`	`a=null`	26
Constructor Calls	`a=new A()`	`a=null`	446
Compound Operator	`a+=1`	`a-=1`	90
Change Numeric Literal	`if(x<0)`	`if(x<1)`	988
Change Boolean Literal	`true`	`false`	225
Change Unary Operator	`!ismethod()`	`ismethod()`	13
Add Not Operator	`if(b)`	`if(!b)`	54
More Specific If	`a&&b`	a,b,`a||b`	222
Less Specific If	`a||b`	a,b,`a&&b`	180
Break and Continue	`break;`	`continue;`	74
Null Assignment	`Object x=y;`	`Object x=null;`	83
Empty Assignment	`s=a.toString();`	`s="";`	91
Primitive Assignment	`int a=b;`	`int a=0;`	232
Change Throw Statement	`throw new A;`	`throw new B;`	132

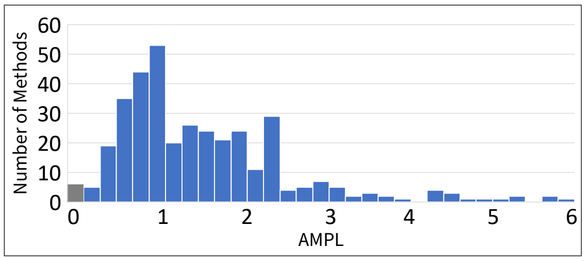


Fig. 5: Histogram of *AMPL* values of the experimental target

Step B: Exclude groups where no differences in *SBFL-Scores* from the subject of consideration.

There are some groups that have methods with the same *SBFL-Scores*, even though the program structures differ within the groups. Those groups are not necessary to analyze factors contributing to higher *SBFL-Scores*. We exclude those groups from Step C.

Step C: Classify factors contributing to higher *SBFL-Scores* and analyze the reasons.

We visually inspect all methods in each group and record the differences in program structure. Then, we classify the differences in program structure into several structure groups. Next, we check the significance of the difference in *SBFL-Scores* for each classified structure group by using a statistical approach. We use Wilcoxon's signed-rank test [17] because *SBFL-Scores* do not follow a normal distribution. Finally, we visually inspect the execution path information for each mutant. By comparing the *SBFL-Scores* and execution path information among the methods, we analyze the reasons why the factors contribute to higher *SBFL-Scores*.

IV. Results

A. The increase in mutants

Fig. 5 shows the Histogram of *AMPL* values of the experimental target. Out of the 365 methods, 359 methods have *AMPL* values greater than 0. The "Times" column in Table II shows the number of times each additional mutant is applied. According to these results, it is considered that the number of mutants increased overall because of additional mutation operators.

B. Factors contributing to higher SBFL-Scores

Out of the 131 groups, 119 groups have differences in *SBFL-Scores* in the same group.

There are 12 groups that have no differences in *SBFL-Scores*. The reasons are as follows:

- There are no conditional expressions in any methods in the group.
- Only the difference between the methods in the same group is that multiple conditions of `if` statements appear in a different order.

By inspecting the 119 method pairs manually, we revealed the following four new factors contributing to higher *SBFL-Scores*:

Factor 1: Using control statements instead of ternary operators or lambda expressions.
Factor 2: Using additional `if` statement for applying *Early Return* [18].
Factor 3: Using multiple `if` statements instead of an `if` statement with a logical operator.
Factor 4: No possibility that exceptions occur in the first statement.

There are some groups that satisfy the factor discovered in the previous research, which is having a low number of statements in the same nesting level.

The factors contributing to higher *SBFL-Scores* and the number of groups that satisfy the factors are listed in Table III.

C. Significance of the difference in SBFL-Scores

Figure 6 (a)-(f) shows box-and-whisker plots with individual data points showing the differences in *SBFL-Scores* between the presence and the absence of each of the five factors, including the four newly discovered factors and one factor discovered in the previous research. Lines between points indicate the difference of *SBFL-Scores* in the same group. To check the significance of the differences in *SBFL-Score*, we used Wilcoxon's signed-rank test and calculated p-values. As can be seen from Figure 6, for factors other than Factor 4, which has a small number of samples, the difference in *SBFL-Scores* between the presence and the absence of the factor was significant ($p < 0.05$).

D. Factors contributing to higher SBFL-Scores

We explain why each factor contributes to higher *SBFL-Scores* with examples below.

Factor 1: Using control statements instead of ternary operators or lambda expressions.

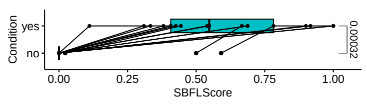
There are several ways to write conditionals and loops.

For example, in Figure 7, method (b) includes an `if` statement. As a result, there are differences in the execution paths when running a test suite, leading to a non-zero *SBFL-Score*. On the other hand, method (a) contains a ternary operator and does not contain any control statements. In such cases, there are no variations in the suspiciousness of each statement. Consequently, it leads *SBFL-Score* of 0. Therefore, by using control statements instead of ternary operators or lambda expressions, *SBFL-Scores* are improved.

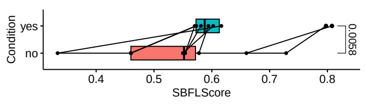
Factor 2: Using additional `if` statement for applying *Early Return*.

TABLE III: The number of groups classified as each factor

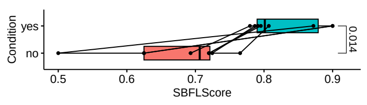
Factor	Number of Groups
Factor 1	20
Factor 2	8
Factor 3	11
Factor 4	4
Factor discovered in the previous research	63
Unknown	13



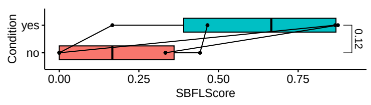
(a) Factor 1



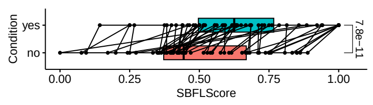
(b) Factor 2



(c) Factor 3



(d) Factor 4



(e) Factor discovered in the previous research

Fig. 6: Distribution of the differences in *SBFL-Score*s between the presence and the absence of each factor

```
return a > b ? 1 : 0;
```

(a)

```
if(a > b)
  return 1;
else
  return 0;
```

(b)

Fig. 7: A method pair indicating the difference between the presence and absence of control statement

```
int s = 0;
for (int i = 0; i < L; i++)
  s += p[i];
return s;
```

(a)

```
if (L < 1)
  return 0;
int s = 0;
for (int i = 0;  i < L; i++)
  s += p[i];
return s;
```

(b)

Fig. 8: A method pair indicating the difference between the presence and absence of *Early Return*

	Program (Input: p)	t_1	t_2	t_3	susp	$rScore$
s_1	`int s = 0;`	✓	✓	✓	1.00	0.33
s_2	`for (int i=0; i<L; i++)`	✓	✓	✓	1.00	0.33
s_3	`s -= p[i]; //bug here`	✓	✓		0.58	0.00
s_4	`return 0;`	✓	✓	✓	1.00	0.00

(a)

	Program (Input: p)	t_1	t_2	t_3	susp	$rScore$
s_1	`int s = 0;`	✓	✓	✓	0.82	0.20
s_2	`if (L < 1)`	✓	✓	✓	0.82	0.20
s_3	`return 0;`			✓	0.00	0.00
s_4	`for (int i=0; i<L; i++)`	✓	✓		1.00	0.60
s_5	`s -= p[i]; //bug here`	✓	✓		1.00	0.60
s_6	`return s;`	✓	✓		1.00	0.60

(b)

Fig. 9: Example of differences in execution path between the presence and absence of Factor 2.

```
if (i < s || i > e)
  return -1;
return 0;
```

(a)

```
if (i < s)
  return -1;
else if (i > e)
  return -1;
return 0;
```

(b)

Fig. 10: A method pair indicating the difference between the presence and absence of a logical OR operator

In some cases, the return value of a method can be determined based on the input before executing all the program statements in the method.

For example, in Figure 8, method (a) contains a `for` loop to process the contents of an array. On the other hand, method (b) also contains an `if` statement to return 0 if the array length is 0.

By using `if` statement to apply *Early Return*, some test cases execute the newly added `if` statement and do not reach `for` loop. Therefore, if a faulty statement exists in `for` loop, the number of successful tests that execute faulty statement decreases. This leads to an increase in the $rScore$ and an increase in the *SBFL-Score* (Figure 9(b)). Therefore, by using `if` statement to apply *Early Return*, *SBFL-Scores* are improved.

Factor 3: Using multiple `if` statements instead of an `if` statement with a logical operator.

There are two ways to write a conditional branch with multiple conditional expressions: connecting conditional expressions with the logical operator or dividing conditional expressions into separate `if` statements.

For example, in Figure 10, method (a) contains a single `if` statement with two conditional expressions connected by the logical OR operator (`||`). On the other hand, method (b) contains two `if` statements with a single condition expression.

Even if there is a fault in the right side of conditional expressions connected by the logical OR operator, a test case that checks the left side of conditional expression becomes a successful test if the left side of conditional expression is correct. Thus, the statement is executed frequently in successful tests. As a result, the suspiciousness of the faulty statement decreases, leading to an increase in the ranking of suspiciousness for other statements that are not faulty (Figure 11(a)). This leads to a decrease in $rScore$ and a decrease in the *SBFL-Score*. Therefore, by separating condition expressions instead of connecting them using logical operators, *SBFL-Scores* are improved (Figure 11(b)).

Factor 4: No possibility that exceptions occur in the first statement.

There are test cases that expect the occurrence of an exception [19]. These test cases may be used to execute SBFL. The *SBFL-Score* measurement tool used in this study

	Program (Input: s,e)	t_1	t_2	t_3	t_4	susp	$rScore$
s_1	`int i = 0;`	✓	✓	✓	✓	0.71	0.66
s_2	`if (i < s || i <= e) // bug here`	✓	✓	✓	✓	0.71	0.66
s_3	` return -1;`	✓		✓	✓	0.47	0.00
s_4	`return 0;`		✓			0.50	0.33

(a)

	Program (Input: s,e)	t_1	t_2	t_3	t_4	susp	$rScore$
s_1	`int i = 0;`	✓	✓	✓	✓	0.71	0.60
s_2	`if (i < s)`	✓	✓	✓	✓	0.71	0.60
s_3	` return -1;`			✓	✓	0.00	0.00
s_4	`else if (i <= e) // bug here`	✓	✓			1.00	1.00
s_5	` return -1;`	✓				0.50	0.20
s_6	`return 0;`		✓			0.50	0.20

(b)

Fig. 11: Example of differences in execution path between the presence and absence of Factor 3.

```
1 String t(String s, Object... args) {
2   StringBuilder b = new StringBuilder(s.length() + 16 * args.length
      );//Exception occurs when s=null
3   ...
4 }
```

(a)

```
1 String t(String s, Object... args) {
2   s = String.valueOf(s);
3   StringBuilder b = new StringBuilder(s.length() + 16 * args.length
      );//Exception occurs when s=null
4   ...
5 }
```

(b)

Fig. 12: A method pair indicating the difference of the location of statements where the exception occurs

	Program (Input: s,t)	t_1	t_2	t_3	susp	$rScore$
s_1	`StringBuilder b = new StringBuilder( s.length() + 16 * args.length);`		✓	✓	1.00	0.00
s_2	`int templateStart = 1; //bug here`		✓	✓	1.00	0.00
...	`...`		...	...	...	...
s_7	` return b.toString();`		✓	✓	1.00	0.00

(a)

	Program (Input: s,t)	t_1	t_2	t_3	susp	$rScore$
s_1	`s = String.valueOf(s);`	✓	✓	✓	0.87	0.00
s_2	`StringBuilder b = new StringBuilder( s.length() + 16 * args.length);`		✓	✓	1.00	0.14
s_3	`int templateStart = 1; //bug here`		✓	✓	1.00	0.14
...	`...`		...	...	...	...
s_8	` return b.toString();`		✓	✓	1.00	0.14

(b)

Fig. 13: Example of differences in execution path between the presence and absence of Factor 4.

determines that when an exception occurs, the statement where the exception occurs is treated as unexecuted. Thus, this specification affects *SBFL-Scores*.

For example, assume the case of the two methods shown in Figure 12. In Method (a), if the input is `s=null`, an exception occurs in line 2. Thus, when running test cases that expect an exception, it is determined that none of the statements in the method are executed. On the other hand, in method (b), if the input is `s=null`, an exception occurs in line 3. Thus, when running test cases that expect an exception, it is determined that line 2 is executed.

If there is a fault after line 3, the test case providing `s=null` always results in an exception occurring on line 3. In this case, test cases expecting an exception become successful tests. As a result, the number of successful tests executing a faulty statement decreases (Figure 13(b)). This leads to an increase in $rScore$ and an increase in *SBFL-Score*.

V. Threat to Validity

A. Internal threats

There are some kinds of granularities, such as statements and code blocks, used in SBFL [20]. In this study, we used statements as a granularity of SBFL. However, if we use another granularity, there is a possibility of producing different results.

Measurement results of *SBFL-Scores* are influenced by the choice of test suites and a mutant generator. Thus, using different test suites and a mutant generator can lead to variations in *SBFL-Scores* and produce different results.

The code coverage tool used in this study determines that when an exception occurs, the statement where the exception occurs is treated as unexecuted. If a different tool is used for collecting code coverage information, the statement where the exception occurs might be treated as unexecuted, potentially negating the experimental results, in particular Factor 4 in Section IV-B.

B. External threats

In this research, we used a dataset created by Higo et al. [7]. This dataset includes small programs containing a single method. If we use programs that have different characteristics from the ones used in this study, such as large programs containing multiple methods, there is a possibility of producing different results.

As described in Section III-A, we set four conditions for the experimental target and excluded programs that did not meet these conditions. As a result, approximately half of the programs were excluded from the experimental targets. Thus, there is a lack of generalizability in the experimental results.

VI. Related Work

SBFL is one of the most popular techniques in fault localization and it has been actively studied in recent years [20], [21].

In this Research, we revealed four program structures to improve the accuracy of SBFL. There is some research about the quality including the accuracy of SBFL. Abrew et al. researched some influences on the accuracy of SBFL [22]. They showed that Ochiai's formula consistently outperforms the other formulas. Furthermore, they showed that a limited number of failing tests is optimal, and additional failing test cases do not affect the accuracy of SBFL. Golagha et al. introduced a technique of predicting the quality of SBFL by using 70 static, dynamic, test suite, and fault-related metrics [23]. Their study showed that it is not necessary to execute all test cases before applying SBFL. However, they also found that applying SBFL right after the first failed test is less effective than applying it after executing all tests for multi-location bugs, which is contrary to the single-location bug study.

There is also some research about improving the accuracy of SBFL. Zhang et al. introduced a technique of improving the accuracy of SBFL by using the PageRank algorithm [24]. The experimental results of this research demonstrate that their technique can outperform state-of-the-art SBFL techniques significantly. Hongdou He et al. introduced a technique for improving the accuracy of SBFL by using Fault Influence Propagation [25]. They conducted an experiment on the real-world fault dataset Defects4J [26] with 33 raw spectrum-based fault locators, which proves that the proposed approach improves the baseline 14.9% average at the top-5 position.

VII. Conclusions and Future Work

In this study, we addressed the shortcomings of previous research [6], namely the small number of programs investigated and the small types of mutation operators. We conducted experiments to measure *SBFL-Score*s and investigated program structures that are suitable for SBFL. As a result of the investigation, we identified four new program structures that are suitable for SBFL.

There are two future challenges below:

Development of an Automated Program Transformation Tool

Both the previous research [6] and our study have identified program structures with high *SBFL-Suitability*. Thus, it is possible to develop a tool that automatically transforms programs to improve their *SBFL-Suitability*. Furthermore, by using the developed tool, it is possible to conduct an experiment to determine if the accuracy of fault localization improves when applying the transformations to programs with existing defects.

Investigation of the Relationship between *SBFL-Suitability* and Other Quality Characteristics

While improved *SBFL-Suitability*, there may be a decrease in maintainability. Actually, the program in Figure 11(b) that is suitable for SBFL contains code clones (Line 2 and Line 4). Therefore, it is a significant challenge to investigate a relationship between *SBFL-Suitability* and other quality characteristics, such as maintainability.

Acknowledgement

This research was supported by JSPS KAKENHI Japan (JP21K18302, JP21H04877, JP22H03567, JP22K11985).

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, *Quantify the time and cost saved using reversible debuggers.* Technical report, Cambridge Judge Business School, 2012.

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

[4] F. Keller, L. Grunske, S. Heiden, A. Filieri, A. van Hoorn, and D. Lo, "A critical evaluation of spectrum-based fault localization techniques on a large-scale software system," in *Proc. IEEE International Conference on Software Quality, Reliability and Security*, 2017, pp. 114–125.

[5] B. Vancsics, T. Gergely, and A. Beszedes, "Simulating the effect of test flakiness on fault localization effectiveness," in *Proc. IEEE Workshop on Validation, Analysis and Evolution of Software Tests*, 2020, pp. 28–35.

[6] Y. Sasaki, Y. Higo, S. Matsumoto, and S. Kusumoto, "Sbfl-suitability: A software characteristic for fault localization," in *Proc. IEEE International Conference on Software Maintenance and Evolution*, 2020, pp. 702–706.

[7] Y. Higo, S. Matsumoto, S. Kusumoto, and K. Yasuda, "Constructing dataset of functionally equivalent java methods," in *Proc. the International Conference on Mining Software Repositories*, 2022, pp. 682–686.

[8] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An Evaluation of Similarity Coefficients for Software Fault Localization," in *Proc. Pacific Rim International Symposium on Dependable Computing*, 2006, pp. 39–46.

[9] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A Practical Evaluation of Spectrum-based Fault Localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[10] 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.

[11] F. Martin, *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[12] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: A Practical Mutation Testing Tool for Java (Demo)," in *Proc. International Symposium on Software Testing and Analysis*, 2016, pp. 449–452.

[13] J. Xuan and M. Monperrus, "Test case purification for improving fault localization," in *Proc. International Symposium on Foundations of Software Engineering*, 2014, pp. 52–63.

[14] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," in *Proc. Symposium and the European Conference on Foundations of Software Engineering*, 2011, pp. 416–419.

[15] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An Empirical Analysis of Flaky Tests," in *Proc. International Symposium on Foundations of Software Engineering*, 2014, pp. 643–653.

[16] Y. Higo, S. Matsumoto, R. Arima, A. Tanikado, K. Naitou, J. Matsumoto, Y. Tomida, and S. Kusumoto, "kgenprog: A high-performance, high-extensibility and high-portability apr system," in *Proc. the Asia-Pacific Software Engineering Conference*, 2018, pp. 697–698.

[17] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in Statistics: Methodology and Distribution.* Springer, 1992, pp. 196–202.

[18] M. A. Saca, "Refactoring improving the design of existing code," in *Proc. IEEE Central America and Panama Convention*, 2017, pp. 1–3.

[19] F. Dalton, M. Ribeiro, G. Pinto, L. Fernandes, R. Gheyi, and B. Fonseca, "Is exceptional behavior testing an exception? an empirical assessment using java automated tests," in *Proc. Int'l Conf. on Evaluation and Assessment in Software Engineering*, 2020, pp. 170–179.

[20] Q. I. Sarhan and A. Beszedes, "A survey of challenges in spectrum-based software fault localization," *IEEE Access*, vol. 10, pp. 10 618–10 639, 2022.

[21] H. A. de Souza, M. L. Chaim, and F. Kon, "Spectrum-based software fault localization: A survey of techniques, advances, and challenges," *arXiv preprint arXiv:1607.04347*, 2017.

[22] R. Abreu, P. Zoeteweij, and A. J. van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques*, 2007, pp. 89–98.

[23] M. Golagha, A. Pretschner, and L. C. Briand, "Can we predict the quality of spectrum-based fault localization?" in *IEEE International Conference on Software Testing, Validation and Verification*, 2020, pp. 4–15.

[24] M. Zhang, X. Li, L. Zhang, and S. Khurshid, "Boosting spectrum-based fault localization using pagerank," in *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis.* Association for Computing Machinery, 2017, pp. 261–272.

[25] H. He, J. Ren, G. Zhao, and H. He, "Enhancing spectrum-based fault localization using fault influence propagation," *IEEE Access*, vol. 8, pp. 18 497–18 513, 2020.

[26] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the international symposium on software testing and analysis*, 2014, pp. 437–440.