Do Exceptional Behavior Tests Matter on Spectrum-based Fault Localization?

Haruka Yoshioka[1], Yoshiki Higo[1], Shinsuke Matsumoto[1], Shinji Kusumoto[1], Shinji Itoh[2], Phan Thi Thanh Huyen[2]

[1] Graduate School of Information Science and Technology, Osaka University, Japan
[2] Hitachi, Ltd., Japan

Abstract. Debugging is a heavy task in software development. Computer-assisted debugging is expected to reduce these costs. Spectrum-based Fault Localization (SBFL) is one of the most actively studied computer-assisted debugging techniques. SBFL aims to identify the location of faulty code elements based on the execution paths of tests. Previous research reports that the accuracy of SBFL is affected by test types, such as flaky tests. Our research focuses on exceptional behavior tests to reveal the impact of such tests on SBFL. Since separating exceptional handling from normal control flow enables developers to increase program robustness, we think the execution paths of exceptional behavior tests are different from the ones of normal control flow tests, which means that the differences significantly affect the accuracy of SBFL. In this study, we investigated the accuracy of SBFL on two types of faults: faults that occurred in the real software development process and artificially generated faults. As a result, our study reveals that SBFL tends to be more accurate when all failing tests are exceptional behavior tests than when failing tests include no exceptional behavior tests.

Keywords: Spectrum-based Fault Localization · exceptions · exceptional behavior test · exception handling

1 Introduction

Debugging is a heavy task in software development. Previous research reported that the process of identifying and correcting faults during the software development process represents over half of development costs [17]. Computer-assisted debugging can reduce these costs.

Fault localization is one of the computer-assisted debugging techniques. So far, many fault localization techniques have been proposed [6, 11]. Spectrum-based Fault Localization (SBFL) is one of the most actively studied techniques [18]. SBFL aims to identify the location of faulty code elements based on the execution paths of tests.

Previous research reports that the accuracy of SBFL is affected by test types, such as flaky tests [16]. In our research, we focus on exceptional behavior tests to reveal the impact of such tests on SBFL. According to the previous investigation [4], separating exceptional handling from normal control flow enables developers to increase program robustness. Therefore, we think the execution paths of exceptional behavior tests are different from the ones of normal control flow tests, which means that the difference significantly affects the accuracy of SBFL. In addition, exceptional behavior tests ensure that their software can handle unexpected situations, recover from errors, and continue to function correctly. From this, we think that exceptional behavior tests can reduce the occurrence of faults, and when faults occur, exceptional behavior tests help developers to identify the causes. Therefore, we hypothesize that exceptional behavior tests are more effective for SBFL.

In this study, we investigated the accuracy of SBFL on two types of faults: faults that occurred in real software development processes and artificially generated faults. Our study revealed that SBFL tended to be more accurate when all failing tests were exceptional behavior tests than when failing tests included no exceptional behavior tests. We confirmed that the number of program statements that need to be checked during debugging was reduced by approximately 33% for faults in real software development processes, and by approximately 66% for artificially generated faults in cases where all failing tests were exceptional behavior tests. Therefore, exceptional behavior tests are important to achieve a higher accuracy of SBFL.

Furthermore, we performed a more detailed categorization of exceptional behavior tests based on the type of exceptions encountered: custom exceptions and standard/third-party exceptions. As a result, we confirmed that SBFL was particularly accurate when all failing tests were exceptional behavior tests that examine the occurrence of standard/third-party exceptions.

The main contributions of our study are as follows.

- This is the first study to investigate the impact of exceptional behavior tests on SBFL.
- We confirmed that SBFL tends to be accurate when all failing tests are exceptional behavior tests.
- We found that SBFL tends to be particularly accurate when all failing tests are exceptional behavior tests that examine standard/third-party exceptions.

2 Preliminaries

2.1 Spectrum-based Fault Localization (SBFL)

SBFL performs fault localization based on execution paths. SBFL is based on the idea that program statements executed in failing tests are likely to be faulty and those executed in passing tests are likely to be less faulty. Fig. 1 shows the procedure for SBFL. The input is a faulty program and its tests. First, the program is run through the tests to obtain the pass or fail of each test and its execution path. From these, the suspicion values are calculated. A suspicion value indicates the likelihood that the program statement includes a fault.

There are many formulae for calculating suspicion values. In previous research, Abreu et al. concluded that Ochiai is the superior formula [1]. Eq. (1) shows the definition of Ochiai.

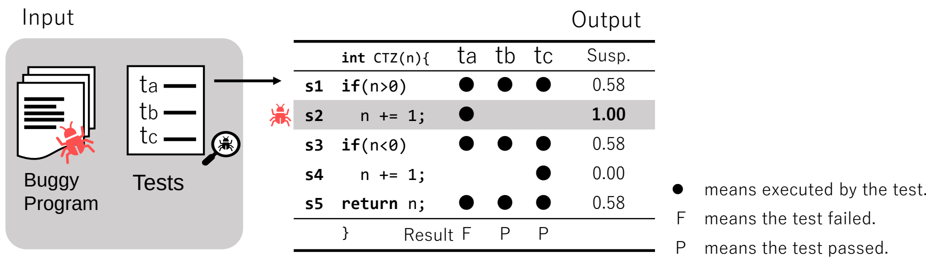


Fig. 1: SBFL flow

```
@Test(expected=ArithmeticException.class)
public void testIncrementToIntegerMaxValue() {
  Math.incrementExact(Integer.MAX_VALUE);
}
```

Fig. 2: Exceptional behavior tests

```
@Test
public void testIncrementExact() {
  int inc = Math.incrementExact(10);
  assertEquals(inc,11);
}
```

Fig. 3: Non-exceptional behavior tests

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

s : a program statement.
$susp(s)$: suspicion value of s.
$totalfails$: the number of failing tests.
$fail(s)$: the number of failing tests that execute s.
$pass(s)$: the number of passing tests that execute s.

2.2 Exceptional Behavior Tests

Exceptional behavior tests verify whether exceptions occur as intended. We investigate a project written in Java. In accordance with previous studies [3], we define exceptional behavior tests as tests that use the methods listed in Table 1.

Fig. 2 shows an example of exceptional behavior test: the test verifies whether an `ArithmeticException` occurs as intended using `expected` attribution in

Table 1: The test frameworks and methods for examining exception handling used in previous research [3].

Framework	Methods for detecting exceptions.
JUnit	Using `assertThrows`.
	Specification of `expected` in @Test.
	Using `ExpectedException`.
TestNG	Specification of `expectedExceptions` in @Test.
AssertJ	Using `assertThatThrownBy`.
	Using `assertThatExceptionOfType`.
	Using `assertThatIOException`.
Common to all frameworks	Using a `fail` call right before a `catch` block.

`@Test` from `JUnit`. The method under test, `Math.incrementExact(int a)`, returns an incremented value of its argument, `int a`. If the increment operation results in an overflow, it throws an `ArithmeticException`. In Fig. 2, `Integer.MAX_VALUE` is specified as the argument of `Math.incrementExact(int a)`. This causes an overflow and triggers the throwing of an `ArithmeticException`.

Exceptions can be classified into custom exceptions and standard/third-party exceptions. Custom exceptions are exceptions implemented by developers themselves, while standard/third-party exceptions are exceptions implemented in standard/third-party libraries. Among exceptional behavior tests, we call tests that inspect custom exceptions as *custom exceptional behavior tests* (hereinafter referred to as `CETest`), and tests that inspect standard/third-party exceptions as *standard/third-party exceptional behavior tests* (hereinafter referred to as `STETest`).

Tests that examine aspects other than exception handling are referred to as non-exceptional behavior tests. All tests other than exceptional behavior tests are non-exceptional behavior tests. An example of a non-exceptional behavior test is shown in Fig. 3. The test provides `10` as an argument to `Math.incrementExact(int a)` and expects the return value to be `11`.

3 Research Questions

In this study, we set the following research questions to investigate whether exceptional behavior tests matter on SBFL.

RQ1: Do exceptional and non-exceptional behavior tests have different effects on SBFL?

We investigate how the ratio of exceptional behavior tests in passing/failing tests affects SBFL. If the ratio significantly affects the accuracy of SBFL, this research enables developers to make preliminary judgments about its reliability.

RQ2: Are there any differences in the length of execution paths between exceptional and non-exceptional behavior tests?

We examine the number of statements executed in exceptional behavior tests and non-exceptional behavior ones. In SBFL, statements executed in failing tests are considered potential candidates for faults. Therefore, the number of statements executed in failing tests is strongly related to the accuracy of the SBFL.

RQ3: Do custom exceptional behavior tests and standard/third party exceptional behavior ones have different effects on SBFL?

We investigate whether `CETests` and `STETests` have different impacts on SBFL. If the accuracy of SBFL differs significantly between these two types of tests, our study can suggest to developers which type of tests they should make proactively.

4 Experimental Setup

4.1 Tools

We use the following tools.

kGenProg[3] kGenProg is an automated program repair tool developed in Higo et al.'s study [5]. We use kGenProg to calculate suspicion values of SBFL.

ExceptionHunter[4] ExceptionHunter is a static analysis tool for Java programs developed in Francisco et al.'s study [3]. ExceptionHunter identifies whether a test is an exceptional behavior test or not.

Mutanerator[5] Mutanerator is a mutant generation tool for Java programs. It applies mutant operators described in Table 2.

4.2 Benchmarks

We take the following benchmarks.

- Faults occurred in the real-world software development process (hereafter referred to as *real faults*).
- Faults artificially generated using Mutanerator (hereafter referred to as *artificial faults*).

We use Defects4J [7] as real faults. Defects4J is a dataset that collects faulty Java programs that occurred in real development processes. Many previous studies use Defects4J as a benchmark [10,12]. Our experiment focuses on six projects within Defects4J: `Math`, `Chart`, `Lang`, `Jsoup`, `JacksonCore`, and `Codec`. kGenProg does not work well with the other projects, so we select these six projects. We exclude some faults due to their inability to be within our environment.

[3] https://github.com/kusumotolab/kGenProg
[4] https://github.com/easy-software-ufal/exceptionhunter
[5] https://github.com/kusumotolab/Mutanerator

Table 2: Mutation operators that are used in Mutanerator.

Mutation operators	Description
Conditional Boundary	Changing the bounds of relational operators.
Increments	Swapping of increment/decrement.
Invert Negatives	Rewriting of negative numbers to positive numbers.
Math	Rewriting arithmetic operators.
Negate Conditionals	Rewriting relational operators.
Void Method Calls	Removing method calls of type void.
Primitive Returns	Rewriting the return value of primitive types to 0.

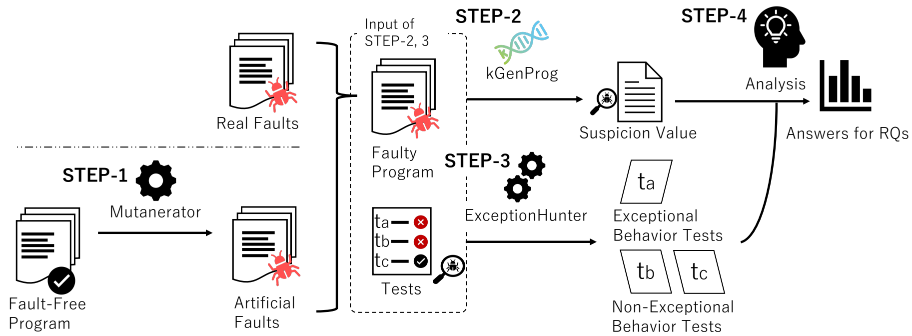


Fig. 4: Workflow of our research

To introduce artificial faults, we apply a two-step process. Initially, we fix faults in Defects4J. Subsequently, we employ Mutanerator to generate new faults. This approach allows us to incorporate artificial faults into our benchmarks.

Artificial faults have been used as benchmarks in previous studies [2, 9] Previous research reported that real and artificial faults have different effects on fault localization effectiveness [14]. However, Yan et al. reported that artificial faults including seeded faults simulate the real scenarios, and may still happen in practice [9]. Therefore, we include artificial faults as a benchmark in this study.

4.3 Workflow

Fig. 4 shows the overall workflow of our experiment. The workflow consists of 4 steps. STEP-1 to STEP-3 are performed automatically by the tools: Mutanerator, kGenProg, and ExceptionHunter. In STEP-4, we manually analyze the impact of exceptional behavior tests on SBFL.

STEP-1. Mutanerator is applied to fault-free programs to generate artificial faults. A fault-free program passes all tests. Mutanerator changes fault-free programs into artificial faults that fail one or more tests.

STEP-2. We input a faulty program and its tests to kGenProg. kGenProg runs the program through its tests and calculates the suspicion values.

STEP-3. ExceptionHunter classifies tests into exceptional behavior tests or not. ExceptionHunter can further classify exceptional behavior tests into `CETest` and `STETest`.

STEP-4. Based on the results of `STEP-2` and `STEP-3`, we investigate whether exceptional behavior tests matter on SBFL.

4.4 Evaluation Metrics

This research uses Rank and rTop-N as evaluation metrics.

Rank

Rank is the position of a faulty statement when arranging program statements in the descending order of suspicion values. If multiple statements share the same suspicion value, Rank takes the average rank of their ties. For a fault with multiple faulty statements, Rank takes the ranking of the first faulty statement, because the localization of the first faulty statement is critical to debugging [10,12,18]. For example, if three faulty statements in a given fault are ranked 2, 5, and 10, Rank of the fault becomes 2.

rTop-N

We create rTop-N with reference to Top-N. Top-N is the number of faults that Rank within N. Top-N is an effective evaluation metric for SBFL and has been used in many previous studies [10,12]. However, Top-N is inappropriate for comparing faults with different sample sizes. For example, there is a difference in meaning between Top-N being 100 out of 200 faults and Top-N being 100 out of 1000 faults. In this study, faults are classified according to the ratio of exceptional behavior tests. Because the number of faults varies after classification, we need to compare faults with different sample sizes. Therefore, we use rTop-N, which is Top-N normalized by the sample size. Eq. (2) shows the definition of rTop-N.

$$\mathsf{rTop\text{-}N} = \frac{\text{The number of faults that } \mathsf{Rank} \text{ is within N.}}{\text{The total number of faults.}} \tag{2}$$

For example, suppose that Rank of two faults are 2 and 10, respectively. In this case, we calculate rTop-5. For the two faults, only the first fault has Rank within 5. Therefore, rTop-5 $= 1/2 = 0.5$.

In this study, we use a value of 5 for N. Previous study [8] reports that 73.58% of developers check only the top 5 elements returned by fault localization techniques.

5 Results and Discussion

We answer RQ1-RQ3 with the experimental results.

5.1 RQ1: Do exceptional and non-exceptional behavior tests have different effects on SBFL?

In RQ1, we investigate whether the ratio of exceptional behavior tests in passing/failing tests affects SBFL. Hereafter, we describe the ratio of exceptional behavior tests in failing tests as `rEFail`, and the one in passing tests as `rEPass`. For example, in Fig. 4, the failing tests are t_a and t_b, and the exceptional behavior tests are t_a. Since the failing exceptional behavior test is only t_a, `rEFail` is $1/2 = 0.5$.

Does rEFail affect SBFL?

First, we investigate the impact of `rEFail` on SBFL. In this experiment, faults are classified as follows.

- **rEFail = 0**
  Failing tests have no exceptional behavior test.
- **0 < rEFail < 1**
  Failing tests have both exceptional and non-exceptional behavior tests.
- **rEFail = 1**
  All failing tests are exceptional behavior tests.

Table 3 shows the number of faults corresponding to each case. The upper part shows real faults and the lower part shows artificial ones. Most of the faults belong to `rEFail` = 0. Regarding real faults, there are only six faults with 0 < `rEFail` < 1 in total. Therefore, for real faults, we compare rTop-5 and Rank only for `rEFail` = 0 and `rEFail` = 1.

First, we examine the effect of `rEFail` on rTop-5. Table 4 shows the results of rTop-5. The hyphenation "—" indicates no fault corresponding to the condition of `rEFail`. The bold letters mean the best rTop-5 for each project. Regarding real faults, only `Math`, `Lang`, and `JacksonCore` have faults with `rEFail` = 1. For all these three projects, the faults with `rEFail` = 1 achieve better rTop-5 than the ones with `rEFail` = 0. As for artificial faults, four projects have faults with `rEFail` = 1: `Math`, `Lang`, `Jsoup`, and `Codec`. Among these projects, `Math`, `Lang`, and `Codec` have the best rTop-5 with `rEFail` = 1. While `Jsoup` shows the worst rTop-5 with `rEFail` = 1, we think this is because there is only one fault with `rEFail` = 1. From the results, we conclude that faults with `rEFail` = 1 tend to achieve better rTop-5 than ones with `rEFail` = 0 or 0 < `rEFail` < 1.

Second, we examine the impact of `rEFail` on Rank. Fig. 5 and Fig. 6 show Rank for real and artificial faults, respectively. The horizontal axis represents the Rank, and each fault's Rank is denoted as a black dot overlaid on the box-and-whisker plot. The outliers of Rank are excluded from the plots to make them easier to read. The red diamond in the figure represents the mean of Rank.

We initially focus on real faults in Fig. 5. Three projects contain `rEFail` = 1: `Math`, `Lang`, and `JacksonCore`. For `Math` and `JacksonCore`, the faults with

Table 3: The number of faults categorized by `rEFail`.

Real faults

	Math	Chart	Lang	Jsoup	JacksonCore	Codec	Total
`rEFail` = 0	67	13	20	14	14	13	141
0 < `rEFail` < 1	1	0	2	1	2	0	6
`rEFail` = 1	12	0	3	0	1	0	16

Artificial faults

	Math	Chart	Lang	Jsoup	JacksonCore	Codec	Total
`rEFail` = 0	399	488	277	166	307	272	1909
0 < `rEFail` < 1	93	0	566	1	33	150	843
`rEFail` = 1	39	0	7	1	0	4	51

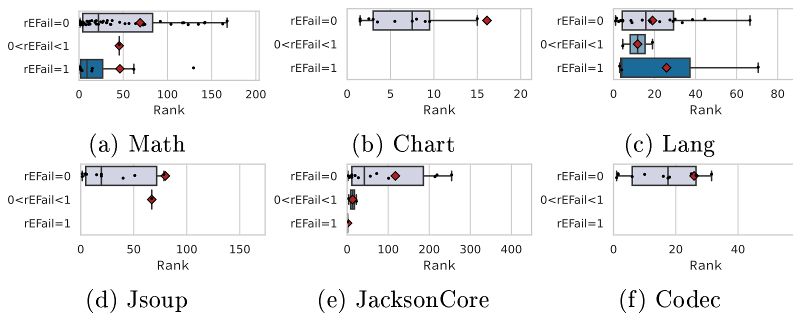


Fig. 5: The distribution of Rank in real faults categorized by `rEFail`.

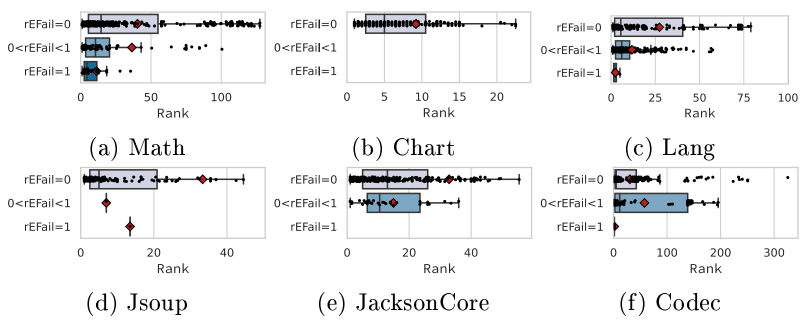


Fig. 6: The distribution of Rank in artificial faults categorized by `rEFail`.

`rEFail` $= 1$ achieve better Rank than the ones with `rEFail` $= 0$ in terms of the first quartile, median, third quartile, and mean values in the box-and-whisker plots. In the case of `Lang`, however, the faults with `rEFail` $= 1$ yield the worse mean and third quartile than those with `rEFail` $= 0$, while showing better first quartile and median values with `rEFail` $= 1$ than with `rEFail` $= 0$. From the results, we conclude that `rEFail` $= 1$ tends to achieve better Rank than `rEFail` $= 0$ for real faults.

Turning our attention to artificial faults in Fig. 6. Four projects have faults with `rEFail` $= 1$: `Math`, `Lang`, `Jsoup`, and `Codec`. Among these projects, `Math`, `Lang`, and `Codec` exhibit the best mean, first quartile, median, and third quartile values when `rEFail` $= 1$. As `Jsoup` does not take the best Rank when `rEFail` $=$ 1, which could be attributed to the fact that `Jsoup` has only one fault with

Table 4: `rEFail` and rTop-5

	Real faults						**Artificial faults**					
	Math	Chart	Lang	Jsoup	JacksonCore	Codec	Math	Chart	Lang	Jsoup	JacksonCore	Codec
`rEFail` $= 0$	0.30	**0.38**	0.35	**0.29**	0.07	**0.23**	0.22	**0.52**	0.49	**0.51**	**0.25**	0.51
$0 <$ `rEFail` < 1	0.00	—	0.50	0.00	0.50	—	0.38	—	0.34	0.00	0.18	0.43
`rEFail` $= 1$	**0.50**	—	**0.67**	—	**1.00**	—	**0.54**	—	**1.00**	0.00	—	**1.00**

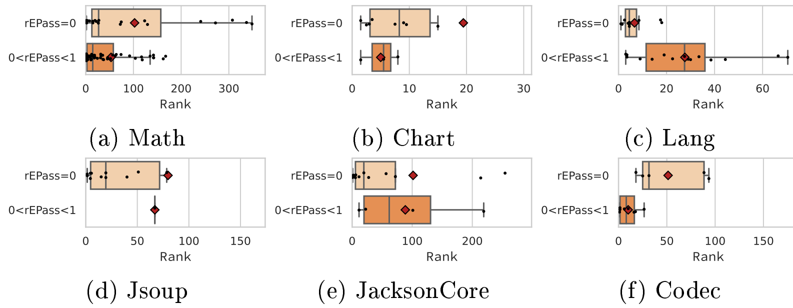
rEFail $= 1$. Overall, these results indicate that rEFail $= 1$ tends to achieve better Rank than rEFail $= 0$ and $0 <$ rEFail < 1 for artificial faults. Then we compare rEFail $= 0$ and $0 <$ rEFail < 1. From Fig. 6, Math, Lang, and JacksonCore have a better Rank with rEFail $= 0$ than with $0 <$ rEFail < 1, while Codec has better Rank when rEFail $= 0$. Therefore, it remains unclear which of rEFail $= 0$ or $0 <$ rEFail < 1 tends to be better.

To confirm our observation, we performed the Mann-Whitney U test at a significance level of 0.01. Since the results of the Shapiro-Wilk test confirmed that the distribution of Rank did not follow a normal distribution for both real and artificial faults, we used the Mann-Whitney U test. First, we focus on real faults. Regarding real faults, we do not distinguish the faults by projects due to the small number of faults with rEFail $= 1$. For real faults, the p-value between rEFail $= 0$ and rEFail $= 1$ is 0.083, which is not statistically significant. We think the lack of statistical significance is likely due to the limited number of faults with rEFail $= 1$. Next, we focus on artificial faults, and Table 5 shows the results. We exclude projects without any faults with rEFail $= 1$ or $0 <$ rEFail < 1 from the table. The bold letters indicate p-values that are below the 0.01 significance level. We focus on the p-values between rEFail $= 0$ and rEFail $= 1$. We confirmed that the p-values for Math and Codec are below the 0.01 significance level. Although there is no significant difference for Jsoup, we think this is because there is only one fault with rEFail $= 1$. As for Lang, while the p-value is not less than the significance level, the mean of Rank with rEFail $= 1$ is 24.93 better than with rEFail $= 0$. Based on these results, we conclude that Rank tends to be better when rEFail $= 1$ than when rEFail $= 0$ for artificial faults. Then we focus on the p-values between $0 <$ rEFail < 1 and rEFail $= 1$. No statistically significant difference is found in Math and Codec, even though they have 39 and 4 faults with rEFail $= 1$, respectively. Therefore, we conclude that it is unclear which Rank tends to be better between rEFail $= 1$ and $0 <$ rEFail < 1.

From these results, we can conclude that SBFL tends to be more accurate when rEFail $= 1$ than when rEFail $= 0$. When comparing rEFail $= 1$ with rEFail $= 0$, the average of Rank with rEFail $= 1$ is about 33% better for real faults and 66% better for artificial faults. We discuss the reason for this in Section 5.2.

Table 5: The p-value in artificial faults

	$0 <$ rEFail < 1 rEFail $= 0$	$0 <$ rEFail < 1 rEFail $= 1$	rEFail $= 0$ rEFail $= 1$
Math	**3.8E-04**	5.6E-02	**1.9E-06**
Lang	5.5E-02	**4.6E-03**	1.8E-02
Jsoup	7.6E-01	1.0	5.3E-01
Codec	5.1E-02	1.5E-02	**2.3E-03**



(a) Math (b) Chart (c) Lang

(d) Jsoup (e) JacksonCore (f) Codec

Fig. 7: The distribution of Rank in real faults categorized by `rEPass`.

Does rEPass affect SBFL?

As in the case of `rEFail`, we classify faults into three categories based on `rEPass`: `rEPass` $= 0$, $0 <$ `rEPass` < 1, and `rEPass` $= 1$. Table 6 shows the distribution of faults for each case. There is no fault with `rEPass` $= 1$, both in real and artificial faults.

Table 7 shows the results of rTop-5. Regarding real faults, while `Math` and `Codec` achieve better rTop-5 with $0 <$ `rEPass` < 1, `Chart`, `Lang`, `Jsoup`, and `JacksonCore` achieve better rTop-5 with `rEPass` $= 0$. For artificial faults, the superiority of rTop-5 with $0 <$ `rEPass` < 1 or `rEPass` $= 0$ varies depending on the projects. Therefore, no clear regularity regarding the impact of `rEPass` on rTop-5 is revealed in this experiment.

Fig. 7 and Fig. 8 show box-and-whisker plots of Rank for real and artificial faults, respectively. Regarding real faults, `Math`, `Chart`, and `Codec` exhibit better Rank when $0 <$ `rEPass` < 1 than when `rEPass` $= 0$. Conversely, for `Lang` and `JacksonCore`, Rank with `rEPass` $= 0$ is distributed in a better range than with $0 <$ `rEPass` < 1. Therefore, it remains uncertain which category yields better Rank for real faults. As for artificial faults, the superiority of either Rank, `rEPass` $= 0$ or $0 <$ `rEPass` < 1, depends on the projects. Consequently, we cannot say which of `rEPass` $= 0$ or $0 <$ `rEPass` < 1 tends to be better.

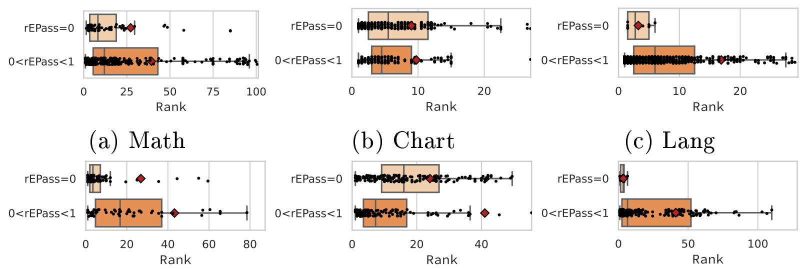
Table 6: The number of faults categorized by `rEPass`.

Real Faults

	Math	Chart	Lang	Jsoup	JacksonCore	Codec	Total
`rEPass` $= 0$	21	11	11	14	16	5	78
$0 <$ `rEPass` < 1	59	2	14	1	1	8	85
`rEPass` $= 1$	0	0	0	0	0	0	0

Artificial faults

	Math	Chart	Lang	Jsoup	JacksonCore	Codec	Total
`rEPass` $= 0$	81	325	8	105	202	13	734
$0 <$ `rEPass` < 1	450	163	842	63	138	413	2069
`rEPass` $= 1$	0	0	0	0	0	0	0



(a) Math (b) Chart (c) Lang
(d) Jsoup (e) JacksonCore (f) Codec

Fig. 8: The distribution of Rank in artificial faults categorized by `rEPass`.

Answer for RQ1: Our findings indicate that when all failing tests are exceptional behavior tests, SBFL results are significantly more accurate compared to scenarios where failing tests have no exceptional behavior tests. The number of statements that must be examined before identifying the first faulty location is reduced by approximately 33% for real faults and 66% for artificial faults when all failing tests are exceptional behavior tests. Therefore, we conclude that exceptional behavior tests matter on SBFL.

5.2 RQ2: Are there any differences in the length of execution paths between exceptional and non-exceptional behavior tests?

In RQ2, we focus on `rEFail` since we revealed that `rEFail` significantly affects the accuracy of SBFL in RQ1. Table 8 shows the results. We find that when `rEFail` $= 1$, the number of program statements executed in failing tests is smaller than when `rEFail` $= 0$ for both real and artificial faults. The reason for this can be attributed to how programs handle abnormal processes. When a program encounters an abnormal process, it throws an exception to deal with the abnormal process. The thrown exception is caught in an appropriate lower-layer class during propagating to higher-layer classes. Therefore, exceptional behavior tests tend to examine the behavior of relatively lower-layer classes that throw exceptions. Since lower-layer classes generally invoke a smaller number of statements than higher-layer classes, the number of statements executed in exceptional behavior tests tends to be smaller than in non-exceptional behavior tests.

Table 7: `rEPass` and rTop-5

	Real faults						**Artificial faults**					
	Math	Chart	Lang	Jsoup	JacksonCore	Codec	Math	Chart	Lang	Jsoup	JacksonCore	Codec
`rEPass` $= 0$	0.19	**0.45**	**0.64**	**0.29**	**0.19**	0.00	**0.40**	0.49	**0.88**	**0.63**	0.12	**0.85**
$0 <$ `rEPass` < 1	**0.37**	0.00	0.21	0.00	0.00	**0.38**	0.25	**0.58**	0.39	0.30	**0.43**	0.48
`rEPass` $= 1$	—	—	—	—	—	—	—	—	—	—	—	—

Therefore, it can be inferred that failing tests tend to execute fewer statements when $\texttt{rEFail} = 1$.

As discussed earlier, SBFL considers statements executed in failing tests as potential fault candidates. Consequently, the shorter the execution paths of failing tests, the higher the accuracy achieved by SBFL. Thus, the result of RQ1 can be attributed to the smaller number of statements executed in exceptional behavior tests.

> **Answer for RQ2**: When all failing tests are exceptional behavior tests, the number of program statements executed in failing tests tends to be small. We think this fact gives better SBFL results when rEFail=1, because the number of candidates for faulty locations is relatively small.

5.3 RQ3: Do custom exceptional behavior tests and standard/third party exceptional behavior ones have different effects on SBFL?

In this RQ, we focus on the ratio of `CETests` and `STETests` in failing tests. The ratio of `CETests` in failing tests is denoted as `rCEFail`, and the ratio of `STETests` is denoted as `rSTEFail`. As in RQ1, we categorize the faults based on `rCEFail` and `rSTEFail`. Due to space constraints, we do not show the distribution of Rank as it is in RQ1. Instead, we utilize the mean of Rank, which is denoted as ave(Rank), for our discussions.

Table 9 shows the results of rTop-5 and ave(Rank). The left side of the table shows rTop-5 and the right side shows ave(Rank). rTop5-All and ave(Rank)-All are rTop-5 and ave(Rank) obtained from the entire set of subjects.

First, we focus on the real faults in the upper part of Table 9. The column rTop5-All indicates that rTop-5 with $\texttt{rCEFail} = 1$ is almost the same as $\texttt{rCEFail} = 0$ and $0 < \texttt{rCEFail} < 1$. In addition, the column ave(Rank)-All indicates that the faults with $\texttt{rCEFail} = 1$ yield the worst ave(Rank) compared to the ones in the other categories. On the contrary, faults with $\texttt{rSTEFail} = 1$ achieve better rTop-5 and Rank than the others. Therefore, for real faults, SBFL is particularly accurate when $\texttt{rSTEFail} = 1$. Next, we focus on the artificial faults in the lower part of Table 9. From rTop5-All and ave(Rank)-All, rTop-5 is 0.21 and ave(Rank) is 4.98 better when $\texttt{rSTEFail} = 1$ than when $\texttt{rCEFail} = 1$. For both real and artificial faults, we can conclude that $\texttt{rSTEFail} = 1$ achieves better rTop-5 and ave(Rank) than those with $\texttt{rCEFail} = 1$.

Table 8: `rEFail` and the number of statements executed in failing tests.

	Real faults						Artificial faults					
	Math	Chart	Lang	Jsoup	JacksonCore	Codec	Math	Chart	Lang	Jsoup	JacksonCore	Codec
$\texttt{rEFail} = 0$	252	73	62	477	328	89	327	74	87	400	292	109
$0 < \texttt{rEFail} < 1$	290	—	32	134	373	—	685	—	97	14	1582	159
$\texttt{rEFail} = 1$	112	—	39	—	16	—	150	—	8	17	—	3

The reason why rTop-5 and Rank with $\texttt{rSTEFail} = 1$ is better than $\texttt{rCEFail} = 1$ lies in shorter execution paths of failing tests. As previously described in Section 5.2, the accuracy of SBFL tends to be high when failing tests have shorter execution paths. Specifically, for $\texttt{rSTEFail} = 1$, the length of the execution paths of failing tests is 47.40, whereas, for $\texttt{rCEFail} = 1$, it is significantly longer, at 168.33.

We now discuss why the execution paths of `CETests` are longer than that of `STETests`. One of the reasons developers create custom exceptions is to handle exceptions related to business logic or workflow. In order to raise a custom exception, it is necessary to invoke a program that implements the business logic or workflow, replicating the situation where the custom exception occurs. On the other hand, standard/third-party exceptions may occur more frequently during program developments and can even occur from simple actions, such as improper method calls or referring `null` objects. Therefore, we think that reproducing the situation where custom exceptions occur is more complex than standard/third-party exceptions. Exceptional behavior tests reproduce a situation where an exception occurs to verify that the intended exceptions are appropriately thrown. Therefore, the complexity of reproducing situations leads `CETests` to achieve a higher number of program statements being executed.

> **Answer for RQ3**: When all failing tests are `STETests`, SBFL tends to be more accurate than when all failing tests are `CETests`. Therefore, `CETests` and `STETests` have different effects on SBFL.

6 Threats to Validity

As evaluation metrics, we used Rank and rTop-N based on Top-N. Top-N is widely used in previous studies [10, 12] Other evaluation metrics may yields different results. In addition, we used real and artificial faults as benchmarks. For the real faults, we used only six Defects4J projects. For artificial faults, we used faults generated from the six projects used as real faults. Our analysis is based on

Table 9: `rCEFail`/ `rSTEFail` and rTop-5, ave(Rank)

Real faults	rTop-5							ave(Rank)						
	Math	Chart	Lang	Jsoup	JacksonCore	Codec	rTop5-All	Math	Chart	Lang	Jsoup	JacksonCore	Codec	ave(Rank)-All
$\texttt{rCEFail} = 0$	**0.33**	**0.38**	**0.40**	**0.27**	0.13	**0.23**	0.31	63.97	**16.12**	**19.24**	**78.93**	109.8	**25.88**	55.37
$0 < \texttt{rCEFail} < 1$	0.00	—	—	—	**0.50**	—	**0.33**	**45.50**	—	—	—	**13.25**	—	**24.00**
$\texttt{rCEFail} = 1$	**0.33**	—	—	—	—	—	**0.33**	86.50	—	—	—	—	—	86.50
$\texttt{rSTEFail} = 0$	0.30	**0.38**	0.35	**0.29**	0.13	**0.23**	0.29	70.58	**16.12**	19.00	79.79	104.5	**25.88**	59.51
$0 < \texttt{rSTEFail} < 1$	0.00	—	0.50	0.00	—	—	0.25	45.50	—	**11.75**	**67.00**	—	—	34.00
$\texttt{rSTEFail} = 1$	**0.67**	—	**0.67**	—	**1.00**	—	**0.70**	**6.08**	—	25.83	—	**1.00**	—	**11.50**

Artificial faults	rTop-5							ave(Rank)						
	Math	Chart	Lang	Jsoup	JacksonCore	Codec	rTop5-All	Math	Chart	Lang	Jsoup	JacksonCore	Codec	ave(Rank)-All
$\texttt{rCEFail} = 0$	0.23	**0.52**	**0.39**	**0.51**	**0.26**	0.53	0.49	38.88	**9.23**	**16.75**	**33.08**	32.24	29.38	23.42
$0 < \texttt{rCEFail} < 1$	0.37	—	—	—	0.13	0.39	0.36	42.92	—	—	—	**16.17**	62.96	51.99
$\texttt{rCEFail} = 1$	**0.54**	—	—	—	—	**1.00**	**0.59**	**11.28**	—	—	—	—	**1.50**	**10.33**
$\texttt{rSTEFail} = 0$	0.27	**0.52**	0.49	**0.51**	**0.26**	0.48	0.52	38.62	**9.23**	27.36	33.35	32.65	41.07	29.76
$0 < \texttt{rSTEFail} < 1$	0.39	—	0.34	0.00	0.13	**0.85**	0.34	**9.25**	—	11.74	**7.00**	**15.75**	**3.23**	11.67
$\texttt{rSTEFail} = 1$	**0.50**	—	**1.00**	0.00	—	—	**0.80**	11.50	—	**2.43**	13.50	—	—	**5.35**

projects as the unit of analysis, and six projects is a very small number. Larger scale experiments may yield different results.

7 Related Works

Francisco et al. surveyed Java projects to assess the prevalence of exceptional behavior tests [3]. Their results showed that approximately 60.91% of projects have at least one test method that examines the behavior of exceptions, and the percentage of exceptional behavior tests is less than 10% in 76.02% of projects. This study also revealed a tendency among developers to prioritize testing for custom exceptions over standard/third-party exceptions. They reported that more focus should be placed on creating exceptional behavior tests. We think the results of our research motivate developers to make exceptional behavior tests.

8 Conclusion

We examined the impact of exceptional behavior tests on SBFL. Our experiments revealed that SBFL was able to localize faulty code elements more accurately when all failing tests were exceptional behavior tests than when failing tests did not include any exceptional behavior tests. Therefore, we concluded that exceptional behavior tests matter on SBFL. In addition, we examined whether the ratio of `CETests` or `STETests` in the failing tests affects SBFL, and found that SBFL was particularly accurate when all failing tests were `STETests`. The results of our study enable developers to make a preliminary assessment of the reliability of SBFL, which is expected to improve the efficiency of debugging.

SBFL is also a technique used in Automated Program Repair (APR) [5] [13] [19]. Previous research have shown that fault localization techniques affect the effectiveness of APR [15]. Therefore, future research includes an investigation of the effect of exceptional behavior tests on APR.

Acknowledgements This research was supported by JSPS KAKENHI Japan (JP20H04166, JP21K18302, JP21K11829, JP21H04877, JP22H03567, JP22K11985)

References

1. Abreu, R., Zoeteweij, P., Golsteijn, R., van Gemund, A.J.: A practical evaluation of spectrum-based fault localization. J. of Systems and Software **82**(11), 1780–1792 (2009)
2. Ali, S., Andrews, J.H., Dhandapani, T., Wang, W.: Evaluating the accuracy of fault localization techniques. In: Proc. Int'l Conf. on Automated Software Engineering. pp. 76–87 (2009)

3. Dalton, F., Ribeiro, M., Pinto, G., Fernandes, L., Gheyi, R., Fonseca, B.: 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. pp. 170–179 (2020)
4. Garcia, A.F., Rubira, C.M., Romanovsky, A., Xu, J.: A comparative study of exception handling mechanisms for building dependable object-oriented software. J. of Systems and Software **59**(2), 197–222 (2001)
5. Higo, Y., Matsumoto, S., Arima, R., Tanikado, A., Naitou, K., Matsumoto, J., Tomida, Y., Kusumoto, S.: kGenProg: A high-performance, high-extensibility and high-portability apr system. In: Proc. Asia-Pacific Software Engineering Conf. pp. 697–698 (2018)
6. Jin, W., Orso, A.: BugRedux: Reproducing field failures for in-house debugging. In: Proc. Int'l Conf. on Software Engineering. pp. 474–484 (2012)
7. Just, R., Jalali, D., Ernst, M.D.: Defects4j: A database of existing faults to enable controlled testing studies for java programs. In: Proc. Int'l Symposium on Software Testing and Analysis. pp. 437–440 (2014)
8. Kochhar, P.S., Xia, X., Lo, D., Li, S.: Practitioners' expectations on automated fault localization. In: Proc. Int'l Symposium on Software Testing and Analysis. pp. 165–176 (2016)
9. Lei, Y., Xie, H., Zhang, T., Yan, M., Xu, Z., Sun, C.: Feature-fl: Feature-based fault localization. IEEE Transactions on Reliability **71**(1), 264–283 (2022)
10. Li, X., Li, W., Zhang, Y., Zhang, L.: Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In: Proc. Int'l Symposium on Software Testing and Analysis. pp. 169–180 (2019)
11. Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S.P.: SOBER: Statistical model-based bug localization. In: Proc. European Software Engineering Conf. Held Jointly with Int'l Symposium on Foundations of Software Engineering. pp. 286–295 (2005)
12. Lou, Y., Ghanbari, A., Li, X., Zhang, L., Zhang, H., Hao, D., Zhang, L.: Can automated program repair refine fault localization? a unified debugging approach. In: Proc. Int'l Symposium on Software Testing and Analysis. pp. 75–87 (2020)
13. Martinez, M., Martin, M.: Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In: Proc. Int'l Symposium on Search Based Software Engineering. pp. 65–86 (2017)
14. Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D., Keller, B.: Evaluating and improving fault localization. In: Proc. Int'l Conf. on Software Engineering. pp. 609–620 (2017)
15. Qi, Y., Mao, X., Lei, Y., Wang, C.: Using automated program repair for evaluating the effectiveness of fault localization techniques. In: Proc. Int'l Symposium on Software Testing and Analysis. pp. 191–201 (2013)
16. Qin, Y., Wang, S., Liu, K., Mao, X., Bissyandé, T.F.: On the impact of flaky tests in automated program repair. In: Int'l Conf. on Software Analysis, Evolution and Reengineering. pp. 295–306 (2021)
17. Tassey, G.: The economic impacts of inadequate infrastructure for software testing (2002)
18. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Trans. on Software Engineering **42**(8), 707–740 (2016)
19. Yuan, Y., Banzhaf, W.: Arja: Automated repair of java programs via multi-objective genetic programming. Transactions on Software Engineering **46**(10), 1040–1067 (2020)