Threats to Validity in Experimenting Mutation-Based Fault Localization

Juyoung Jeon
Handong Global University
Pohang, South Korea
juyoungjeon@handong.edu

Shin Hong
Handong Global University
Pohang, South Korea
hongshin@handong.edu

ABSTRACT

Mutation-based fault localization (MBFL) is a promising direction toward improving fault localization accuracy by leveraging dynamic information extracted by mutation testing on a target program. One issue in investigating MBFL techniques is that experimental evaluations are prone to various validity threats because there are many factors to control in generating and running mutants for fault localization. To understand different validity threats in experimenting MBFL techniques, this paper reports our re-production of the MBFL assessments with Defects4J originally studied by Pearson et al. Having the JFreeChart artifacts of Defects4J (total 26 bug cases) as study objects, we conducted the same empirical evaluation on two MBFL (Metallaxis and MUSE) and six SBFL techniques (DStar, Op2, Ochiai, Jaccard, Barniel, Tarantula) while identifying and managing validity threats in alternative ways. As results, we found that the evaluation on the studied techniques change in many parts from the original results, thus, the identified validity threats should be managed carefully at designing and conducting empirical assessments on MBFL techniques.

ACM Reference Format:
Juyoung Jeon and Shin Hong. 2020. Threats to Validity in Experimenting Mutation-Based Fault Localization. In *New Ideas and Emerging Results (ICSE-NIER'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3377816.3381746

1 INTRODUCTION

Fault localization is to identify buggy locations in target program code, whose operation is suspected to cause failures in a target program execution. Accurate localization of a fault is crucial to the successes of debugging, not only for human developers but also for automated program repair techniques [3].

To achieve accurate and fully automated fault localization, statistical fault localization techniques utilize static and dynamic information on a target program to indicate code locations having high correlations with failures as buggy/suspicious. For last two decades, extensive research efforts have been put toward improving fault localization techniques by leveraging more information on target programs and enhancing statistical analysis schemes. This effort is on going to answer strong demand on high accuracy of fault localization. For instance, the learning-to-rank approaches [2, 4] is one of recent directions that possibly lead to new breakthrough.

Mutation-based fault localization (MBFL) [1, 2, 5, 6] is a recent direction to utilize mutation testing in fault localization. MBFL techniques first generate a number of variant programs (i.e., mutants), and run them with test cases to measure how test case execution results changes when a code element is mutated/alternated. Using this information, MBFL techniques statistically infer code locations highly relevant to the occurrence of failures. The initial works show promising results that MBFL techniques produce highly accurate results, even with large and complex target programs [1].

To advance fault localization techniques, it is important to conduct empirical evaluation systematically while mitigating validity threats as much as possible. One issue in evaluating and assessing MBFL techniques is that there are many factors to control, compared to spectrum-based fault localization (SBFL) techniques. Unlike SBFL which receives the dynamic information from a target program executions, MBFL generates dynamic information by injecting artificial faults and analyzing how a mutant execution changes the target program behaviors. For instance, even for similar MBFL techniques, the mutation testing settings are quite different from empirical study to study. Thus, there are relatively more complicated, non-trivial steps to be made, therefore, depending on how these steps are controlled, the end result may change significantly.

Pearson et al. [7] presents quantitative evaluations of well-known MBFL and SBFL techniques with Defects4J. The Defects4J benchmark provides a large number of various real buggy versions of well-known open-source Java programs. By providing comprehensive and real-world study objects (total 395 real bug cases), Defects4J enables the fault localization research community to conduct empirical assessments while effectively managing the external validity threat of having a limited number of study objects. Pearson et al. [7] aims to conduct experiments on the studied SBFL and MBFL techniques under a uniform framework, with extensive study objects, in order to complement previous research works which compare them with different experiment designs.

In this study, we conducted empirical assessments of MBFL and SBFL techniques for the Defects4J artifacts, while using different configurations of controlled variables to Pearson et al. [7]. We found that the criteria/assumption for controlling factors of mutation testing are notably different in Pearson et al. [7] and the presented MBFL works such as Metallaxis [6], MUSE [5], and MUSEUM [1]. To identify how these factors matter the conclusion of empirical assessments, we re-conducted the empirical evalution of MBFL and SBFL techniques with the 26 bug cases of JFreeChart in Defects4J with an alterantive experiment set up. We configured these mutation testing phase as consistent with the earlier MBFL

ICSE-NIER'20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7126-1/20/05…$15.00
https://doi.org/10.1145/3377816.3381746

Table 1: The revised 26 JFreeChart artifacts in Defects4J

No.	Bug type	Buggy lines	Failing TCs	Passing TCs	Target lines	Mutants
C1	Operator misusage	1 (1)	1* (1)	1299 (427)	749 (13269)	1218 (989)
C2	Omission fault	7 (8)	1* (2)	837 (347)	132 (2515)	212 (198)
C3	Omission fault	3 (1)	1* (1)	569 (48)	145 (1571)	269 (212)
C4	Omission fault	1 (2)	22* (22)	1715 (403)	2637 (46751)	4930 (4072)
C5	Omission fault	4 (2)	1* (1)	302 (125)	52 (524)	110 (80)
C6	Method misusage	1 (1)	2* (2)	947 (477)	130 (1088)	270 (193)
C7	Variable misusage	2 (2)	1* (1)	524 (14)	125 (939)	287 (221)
C9	Incorrect bound check	1 (1)	1 (1)	514 (31)	206 (1930)	536 (567)
C10	Variable misusage	1 (1)	1* (1)	0 (0)	4 (141)	2 (0)
C11	Variable misusage	1 (1)	1* (1)	263 (263)	20 (302)	57 (37)
C12	Operator misusage	1 (1)	1* (1)	1028 (199)	645 (8946)	929 (808)
C13	Variable misusage	1 (1)	1* (1)	479 (138)	282 (1723)	499 (446)
C14	Omission fault	5 (4)	1 (4)	977 (487)	488 (11601)	715 (922)
C15	Omission fault	5 (3)	1 (1)	1053 (196)	1315 (23139)	2025 (1809)
C16	Value misusage	2 (3)	7* (8)	483 (13)	49 (656)	88 (61)
C17	Method misusage	1 (1)	1* (1)	497 (48)	104 (2173)	222 (169)
C18	Incorrect bound check	1 (2)	2* (4)	545 (237)	89 (749)	133 (107)
C19	Omission fault	1 (2)	1* (2)	977 (359)	890 (8467)	1645 (1391)
C20	Variable misusage	1 (1)	1* (1)	658 (295)	101 (969)	105 (61)
C21	Omission fault	3 (4)	1* (1)	581 (12)	116 (1186)	211 (171)
C22	Method misusage	1 (7)	2* (6)	51 (31)	61 (420)	94 (139)
C24	Variable misusage	1 (1)	1* (1)	9 (9)	12 (65)	34 (30)
C25	Omission fault	2 (6)	1 (4)	1148 (5)	3618 (21450)	7104 (5971)
C26	Omission fault	2 (2)	22* (22)	1421 (517)	6268 (39569)	11886 (9162)

works [1, 5, 6]. As a part of this, we carefully reviewed and checked the components of the 26 artifacts of Defects4J, including the target source code, the test cases, the buggy line definition, to identify possible validity threats regarding the Defects4J artifacts, and try best to eliminate errors/noises before the experiments [1].

After generating mutant execution information, we applied the two MBFL and the six SBFL techniques and then evaluated them by comparing fault localiation accuracies. We observed that, for 19 bug cases, the evaluation results of our study is different from that of Pearson et al. [7] (Section 3) as the controlled variables are configured with alternative settings (see Section 4.1). These results imply that the observed threats to validity should be carefully managed at designing, conducting and analyzing MBFL experiments.

2 STUDY DESIGN

2.1 Aspects of MBFL Experiments

2.1.1 Independent and Dependent Variables. Both MBFL and SBFL techniques receive the source code of a target buggy program and the testing results of passing and failing test cases as input, and then these techniques assign suspiciousness scores as real values to the set of source code lines. The testing result of a test case consists of structural coverage as a set of covered code elements (e.g., covered lines, covered statements) and the output results of the test case execution (e.g., stack trace at crash).

MBFL techniques additionally receive the mutation testing results on a target program. MBFL techniques utilize mutation testing tools (e.g., Proteum, Major, PIT) to construct mutation testing results by generating mutants and running the mutants with the given passing and failing test cases. The mutation testing results consist of the information of generated mutants (e.g., mutated line, applied mutation operator), the testing result of a test case execution on each mutant (i.e., either pass or fail), and the output result of each test case execution on a mutant.

For given input, a SBFL or MBFL technique uses a unique scoring formula (scheme) to compute the suspiciousness score of each line of code as a real value. The accuracy of a MBFL technique is determined by the earliest index of a buggy line in a sequence where the lines of source code are sorted in the descending order of suspiciousness scores. Since there may exist multiple lines with the same score, the evaluation uses the minimum, the mean, or the maximum of the possible indices.

Note that empirical evaluation of fault localization techniques compares the accuracies (i.e., dependent variable) of different MBFL and SBFL techniques (i.e., independent variable) when the same input (i.e., control variable) is given to them.

2.1.2 Control variables. Experiments configure the remaining factors as controlled variables in order to mitigate various threats to the validity of the empirical assessments of fault localization techniques. The followings are the criteria to control these factors in the MBFL experiments [1, 5, 6]:

- **Target program.** An experiment controls the properties on a target program such as the domain (functionality), the size (i.e., lines of code) or bug type, to make study objects (i.e., the set of target buggy programs) represent common cases of debugging in the real world. In most cases [1, 5, 6], an experiment restricts that a target buggy program to contain exactly one bug (except for the intended cases of multiple-fault localization).
- **Bug locations.** An experiment defines bug location as a set of executable code locations/lines whose operation contributes to the failure in a failing test case execution. The definition of bug locations must be given from the developer (i.e., domain expert) as ground truth. This information is used for evaluating the accuracy of fault localization techniques.
- **Test cases.** Since the quality and quantity of test cases significantly plays in fault localization performance, experiments control the quality and the quantity of test cases to well represent real-world cases. Most studies employ test cases in the regression test suite written and/or maintained by the developers. An experiment employs the test cases that are likely used by developers in debugging processes. Developers expect test cases to be deterministic (i.e., non-flaky) such that they can reproduce the test execution results by the test input. It is unlikely to use test cases that are determined by unmanagable factors such as timing, hardware status, external circumstances, etc. Moreover, developers expect that a test case execution exercises a single program execution scenario.
- **Mutation testing.** The mutation testing results given to MBFL techniques depend on how a mutation testing tool is configured in an experiment. An experiment controls a set of mutation operators applied for generating mutants, a set of target code locations where mutation operators are applied, and mutant sampling criterias such as the limit of the number of mutant generation on a single code line [1].

2.2 New Experiments with Defects4J

To identify how configuration of the controlled variables affect MBFL results, we reproduced a part of the empirical evaluation by Pearson et al. [7] with a different configuration of the controlled variables. For this study, we chose the whole set of the JFreeChart artifacts (26 bug cases) in Defects4J from total 395 bug cases in Defects4J. By limiting the number of study objects, we were able to conduct more detailed and quantitative analyses on experiment data. Using the bug cases from the same project (i.e., JFreeChart) also limits structural differences among the study objects.

[1] All revised artifacts can be found at https://github.com/arise-handong/icse20-mbfl

Table 2: The experiment results of the eight fault localization techniques with the 24 study objects

No.	Metallaxis		MUSE		DStar		Op2		Ochiai		Jaccard		Barinel		Tarantula	
	Bug-Rank	Tech-Rank	Bug-Rank	Tech-Rank	Bug-Rank	Tech-Rank	Bug-Rank	Tech-Rank	Bug-Rank	Tech-Rank	Bug-Rank	Tech-Rank	Bug-Rank	Tech-Rank	Bug-Rank	Tech-Rank
C1	1.5 (3564.5)	1 (8)	2 (3473.5)	2 (7)	35 (36)	3 (1)	35 (36)	3 (1)	35 (36)	3 (1)	35 (36)	3 (1)	35 (36)	3 (1)	35 (36)	3 (1)
C2	9 (25)	1 (1)	81.5 (599)	8 (8)	36 (92.5)	2 (2)	36 (139.5)	2 (7)	36 (92.5)	2 (2)	36 (92.5)	2 (2)	36 (92.5)	2 (2)	36 (92.5)	2 (2)
C3	6.5 (7)	1 (7)	67 (340)	8 (8)	6.5 (6.5)	1 (1)	6.5 (6.5)	1 (1)	6.5 (6.5)	1 (1)	6.5 (6.5)	1 (1)	6.5 (6.5)	1 (1)	6.5 (6.5)	1 (1)
C4	141 (56)	5 (2)	2325 (25263)	8 (8)	49 (167)	2 (3)	16 (16)	1 (1)	55 (173)	3 (4)	63 (181)	4 (5)	186 (744)	7 (6)	166 (744)	6 (6)
C5	3 (3)	1 (1)	13 (224)	8 (8)	7.5 (7.5)	2 (2)	7.5 (7.5)	2 (2)	7.5 (7.5)	2 (2)	7.5 (7.5)	2 (2)	7.5 (7.5)	2 (2)	7.5 (7.5)	2 (2)
C6	99 (100)	7 (4)	126 (594)	8 (8)	49 (57)	2 (2)	6 (15)	1 (1)	51 (57)	3 (2)	83 (103)	4 (5)	97 (103)	6 (5)	83 (103)	4 (5)
C7	83.5 (677.5)	8 (7)	25 (677.5)	1 (7)	34.5 (72)	2 (1)	34.5 (72)	2 (1)	34.5 (72)	2 (1)	34.5 (72)	2 (1)	34.5 (72)	2 (1)	34.5 (72)	2 (1)
C9	10 (15)	7 (7)	22 (22)	8 (8)	8.5 (8.5)	1 (1)	8.5 (8.5)	1 (1)	8.5 (8.5)	1 (1)	8.5 (8.5)	1 (1)	8.5 (8.5)	1 (1)	8.5 (8.5)	1 (1)
C10	1.5 (30.5)	1 (7)	2.5 (30.5)	2 (7)	2.5 (2.5)	2 (1)	2.5 (2.5)	2 (1)	2.5 (2.5)	2 (1)	2.5 (2.5)	2 (1)	2.5 (2.5)	2 (1)	2.5 (2.5)	2 (1)
C11	8.5 (7.5)	1 (1)	18.5 (95)	8 (8)	10 (10)	2 (2)	10 (10)	2 (2)	10 (10)	2 (2)	10 (10)	2 (2)	10 (10)	2 (2)	10 (10)	2 (2)
C12	405.5 (2517.5)	8 (8)	178 (2457.5)	7 (7)	13.5 (19.5)	1 (1)	13.5 (19.5)	1 (1)	13.5 (19.5)	1 (1)	13.5 (19.5)	1 (1)	13.5 (19.5)	1 (1)	13.5 (19.5)	1 (1)
C13	33.5 (36)	2 (2)	21.5 (6.5)	1 (1)	35 (44.5)	3 (3)	35 (44.5)	3 (3)	35 (44.5)	3 (3)	35 (44.5)	3 (3)	35 (44.5)	3 (3)	35 (44.5)	3 (3)
C14	5 (5)	8 (1)	1.5 (5)	1 (1)	4.5 (25.5)	2 (7)	4.5 (600.5)	2 (8)	4.5 (15.5)	2 (3)	4.5 (15.5)	2 (3)	4.5 (15.5)	2 (3)	4.5 (15.5)	2 (3)
C15	686.5 (6451.5)	7 (8)	940 (6333.5)	8 (7)	32 (34.5)	1 (1)	32 (34.5)	1 (1)	32 (34.5)	1 (1)	32 (34.5)	1 (1)	32 (34.5)	1 (1)	32 (34.5)	1 (1)
C16	31.5 (150)	8 (8)	11.5 (144)	1 (7)	29.5 (1.5)	3 (1)	29.5 (1.5)	3 (1)	29.5 (1.5)	3 (1)	29.5 (1.5)	3 (1)	20.5 (2)	2 (5)	29.5 (2)	3 (5)
C17	2.5 (1.5)	8 (2)	1 (1)	1 (1)	2 (2)	2 (3)	2 (2)	2 (3)	2 (2)	2 (3)	2 (2)	2 (3)	2 (2)	2 (3)	2 (2)	2 (3)
C18	1 (3)	1 (2)	3 (2)	2 (1)	4.5 (4.5)	3 (3)	4.5 (4.5)	3 (3)	4.5 (4.5)	3 (3)	4.5 (4.5)	3 (3)	10 (10.5)	7 (7)	10 (10.5)	7 (7)
C19	535 (3)	8 (1)	272 (2251.5)	7 (8)	3 (3.5)	1 (2)	3 (873.5)	1 (7)	3 (3.5)	1 (2)	3 (3.5)	1 (2)	3 (3.5)	1 (2)	3 (3.5)	1 (2)
C20	54 (737.5)	8 (7)	39 (737.5)	7 (7)	3 (6)	1 (1)	3 (6)	1 (1)	3 (6)	1 (1)	3 (6)	1 (1)	3 (6)	1 (1)	3 (6)	1 (1)
C21	11 (16.5)	1 (7)	27.5 (251.5)	8 (8)	21.5 (10.5)	2 (1)	21.5 (10.5)	2 (1)	21.5 (10.5)	2 (1)	21.5 (10.5)	2 (1)	21.5 (10.5)	2 (1)	21.5 (10.5)	2 (1)
C22	6 (10)	8 (4)	2 (1)	1 (1)	3.5 (47.5)	2 (7)	3.5 (57.5)	2 (8)	3.5 (8.5)	2 (2)	3.5 (8.5)	2 (2)	5 (18.5)	6 (5)	5 (18.5)	6 (5)
C24	2 (2)	2 (1)	1.5 (12)	1 (8)	3 (3)	3 (2)	3 (3)	3 (2)	3 (3)	3 (2)	3 (3)	3 (2)	3 (3)	3 (2)	3 (3)	3 (2)
C25	1072 (444.5)	7 (4)	1167.5 (87)	8 (1)	31.5 (3504.5)	1 (7)	31.5 (3504.5)	1 (7)	31.5 (3290.5)	1 (5)	31.5 (3290.5)	1 (5)	31.5 (127.5)	1 (2)	31.5 (127.5)	1 (2)
C26	7 (8)	2 (2)	4 (4)	1 (1)	137 (140)	4 (3)	137 (140)	4 (3)	137 (140)	4 (3)	137 (140)	4 (3)	105 (738)	3 (7)	495 (738)	8 (7)

We first obtained the original artifacts of the 26 JFreeChar bug cases used for Pearson et al. [7] [2]. And then, we carefully reviewed all elements of each artifact to identify the factors that make the experiments not coherent to the common criteria for configuring the controlled variables (Section 2.1). For such cases, we revised them to manage the potential threats.

Table 1 summarized the results of the reviews and the revisions on the 26 study objects. The first column indicates the name of artifact where 'C1' means `Chart-1` in Defects4J. The details of this process are described in the subsequent subsections.

2.2.1 Clarifying Target Bug. We checked whether all bug locations correspond to the same fault, or multiple faults by running all given failing test cases and examining their root causes manually. In addition, we reviewed the commit (the difference between the buggy and the corrected versions) to check if they are fixing multiple bugs at the same time. The followings are the found issues:

- For `Chart-8`, it is impossible to reproduce a failing test case execution because the failing test case is flaky due to a Timezone dependency. Consequently, we excluded `Chart-8` from the study.
- For `Chart-23`, it is impossible to reproduce a failing test case execution. After careful review, we found that the specified bug locations are not executed by any failing test case and not relevant to the bug description (commit-log). Consequently, we exclude `Chart-23` from the study.
- For seven artifacts (C2, C14, C16, C18, C19, C22 and C25), we found that each of these concerns two or more faults simultaneously. We conjecture that the commits where such an artifact originates address multiple independent issues at the same time. For each of these seven artifacts, we decided to use one arbitrary bug as the target. The second column of Table 1 presents a short description of the target bug of each artifact after all revisions. And then, we revised the artifacts in the following two ways:
  - remove code lines of the other bugs than the target bug from the bug locations, and
  - remove the failing test cases that fail by the other bugs from the failing test cases. The third column of Table 1 shows the number of the failing test cases used for the our study and that of the original study in the parenthesis.

[2] https://bitbucket.org/rjust/fault-localization-data/src/master/

2.2.2 Redefining Bug Locations. Defects4J defines the bug locations of each artifact as the changed or deleted lines from the buggy version to the patched version. We first detected the lines not relevant to the target bug (e.g., non-executable code lines, changes for the other issues) and removed them from the bug location definitions.

For an omission fault (10 out of the 26 study objects), Defects4J defines the bug location as only one line right next to where a new line is added in the corrected version. We found that this definition was too restrictive because adding the new lines fix a bug even when they are inserted to other locations. To resolve this issue, we extended the bug location definition of nine omission faults to include all lines in the same code block where the patch adds a new line to fix the problem. By careful reviews, we confirm that all these lines also fix the bug with the same patches. The third column of Table 1 shows the number of the redefined bug locations (the number of the original bug locations is in the parenthesis).

2.2.3 Purifying Failing Test Cases. We manually purified the failing test cases of 20 among the 24 study objects [9] (marked with asterisk at the fourth column of Table 1) because these failing test cases have multiple test scenarios with multiple assertion statements, such that irrelevant operations are executed in the failing test case execution before the failure occurs. Failing test cases with multiple assertions may accidentally result that mutants fail on other assertions than the one that detects the failure on the original program. To mitigate this threat, we removed irrelevant lines in a failing test case to make each failing test case contain exactly one assertion statement.

2.2.4 Re-collecting Passing Test Cases. Defects4J uses an arbitrary criteria for collecting passing test cases among all test cases given in JFreeChart. To make the experiments more systematic, we re-collected the passing test cases for the experiments by including all passing test cases that cover at least one same line as a failing test case [1]. The fifth column of Table 1 shows the number of passing test cases used in our study together with that of Pearson et al. [7] in the parenthesis. At the end, the number of passing test cases increased by 458 on average.

2.2.5 Confirming All Test Cases Being Deterministic. To detect flaky test cases, we repeated the executions of all selected test cases 5 times while checking whether or not the line coverage change in

any repeat. We observed no chances, thus, we believe that there exists no flakiness in the selected test cases.

2.2.6 Redefining Target Lines. We defined *target lines* as the lines of code to assign suspiciousness scores, which are the set of lines covered by at least one failing test cases. The rest of the code lines are not suspected to be buggy because they are executed only by passing test cases and do not contribute to any observed failures. Thus, we consider them to have a lower suspiciousness score than any of the target lines. The sixth column of Table 1 reports the number of the target lines. The number in a parenthesis is of the original experiment.

2.2.7 Generating Mutants with More Mutation Operators. Pearson et al. [7] employed 72 mutation operators of a Java mutation testing tool Major version 1.2.1. Our study applied total 113 mutation operators of Major 1.3.4 (i.e., latest version at the experiments) to the taret lines to generate mutants. The last column of Table 1 shows the number of the generated mutants. Compared to the original ones (the numbers in the parenthesis), the experiments in our study uses 13.7% more mutants on average.

3 RESULT

Table 2 shows the best ranking of the bug locations ("Bug-Rank" columns) and the ranking of a fault localization techniques among the eight fault localization techniques ("Tech-Rank" columns) for each study object. Note that `Chart-8` and `Chart-23` are excluded because the given failing test cases do not reproduce the intended failures (Section 2.2.1). In the second to the last columns, each cell represents the result of the new experiment, and the number in the parenthesis represents the corresponding result reported in the experiment artifacts of Pearson et al. [7].

The new experiment results show that Metallaxis show the best accuracies among the eight techniques in 8 study objects, and MUSE in 8 study objects. And, for 15 study objects, Metallaxis or MUSE shows better accuracies than all SBFL techniques. For other 8 study objects, at least one of SBFL techniques show better accuracies than Metallaixs and MUSE. For the remaining one study object, Metallaxis and the six SBFL techniques achieve the same best accuracy.

Comparing the new results with the Pearson et al.'s results, the best performing technique has changed nine study objects. And, the ranking among the fault localization techniques has changed in 19 study objects. We can also find that the Bug-Rank significantly reduced, and thus, the accuracies improved in a large degree. For instance, the average Bug-Rank over all techniques and all study objects in the new experiment results is 65.6, meanwhile the average Bug-Rank in Pearson et al. is 420.8. We conjecture that main reasons for this change is that the new experiments make all target lines have a higher suspiciousness scores than all non-target lines.

We observed that the changes of Bug-Rank of the two MBFL techniques (average 1039.9) is much greater than that of the six SBFL techniques (average 127.0). We also found that Metallaxis and MUSE show different results from each other for many cases. In contrast, the SBFL techniques show the same ranking of the bug locations for most of the study objects.

4 DISCUSSION

4.1 Factors of MBFL Result Changes

It was found that, with eight of the 24 experimented study objects, the Tech-Rank numbers of Metallaxis and/or MUSE are notably changed from Pearson et al. For five study objects (C1, C7, C10, C21, C24), the Tech-Rank of Metallaxis and/or MUSE becomes better than the SBFL techniques whereas, for the other three (C4, C19, C25), the Tech-Rank of Metallaxis and/or MUSE becomes worse. We found the factors that contribute most to these changes as follows:

- The accuracies improvements for C1 (MUSE and Metallaxis), C7 (MUSE) and C21 (Metallaxis) are mainly because of using more mutation operators in the new experiments. As the new mutation operators discover useful mutants, MBFL techniques get better in localizaing the bugs.
- The accuracies improvement for C24 (MUSE) are mainly because of testing purification of the failing test cases. This result implies that the MBFL techniques can locate the bugs more accurately when the failing test case generates more clear and accurate test execution results.
- For C10 (MUSE and Metallaxis), the acccuracies are improved as non-target lines are assigned with lowest suspiciousness scores.
- For C4 (Metallaxis), C19 (Metallaxis), and C25 (Metallaxis and MUSE), we found that elimination of irrelevant failing test cases and failing test case purification degrades the accuracies of the MBFL techniques. We conjected that the noise in the original artifacts exaggreate their accuracies.

4.2 Threats to Validity in Using Defects4J

Section 2.2 details our results of reviewing and revising Defects4J to mitigate the effect of noises in the artifacts, and also explore alternative acceptable configuration of MBFL experiments. Section 3 implies that the comparison results among the eight fault localization techniques have changed when the empirical assessments are taken with a different set up. These results demonstrate that the empirical evaluation of fault localization using the Defect4J benchmark should put efforts to manage these identified threats to validity carefully, and also consider their potential influences in interpreting the experiment results. In addition, these results indicate that it would be valuable to have more investigations to understand the characteristics of the Defects4J benchmark (e.g.,[8]) and improve methodologies to utilize these assets more systematically for empirical evaluation.

ACKNOWLEDGEMENT

This research was supported by Next-Generation Information Computing Development Program (2017M3C4A7068179) and Basic Science Research Program (2017R1C1B1008159) through NRF of Korea.

REFERENCES

[1] S. Hong, T. Kwak, B. Lee, Y. Jeon, B. Ko, Y. Kim, and M. Kim. 2017. MUSEUM: Debugging Real-World Multilingual Programs Using Mutation Analysis. *IST* 82 (Feb. 2017), 80–95.
[2] Y. Kim, S. Mun, S. Yoo, and M. Kim. 2019. Precise Learn-to-Rank Fault Localization Using Dynamic and Static Features of Target Programs. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* 28, 4 (Oct. 2019).
[3] L. Kui, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon. 2019. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. In *ICST*.
[4] X. Li, W. Li, Y. Zhang, and L. Zhang. 2019. DeepFL: Integrating Multiple Fault Diagnosis Dimensions for Deep Fault Localization. In *ISSTA*.
[5] S. Moon, Y. Kim, M. Kim, and S. Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *ICST*.
[6] M. Papadakis and Y. Le-Traon. 2015. Metallaxis-FL: mutation-based fault localization. *STVR* 25 (2015), 605–628.
[7] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. 2017. Evaluating and Improving Fault Localization. In *ICSE*.
[8] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. .A. Maia. 2018. Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. In *SANER*.
[9] J. Xuan and M. Monperrus. 2014. Test case purification for improving fault localization. In *FSE*.