Improving Mutation-Based Fault Localization via Mutant Categorization

Xia Li
Department of Software Engineering and
Game Design, Kennesaw State University
xli37@kennesaw.edu

Durga Nagarjuna Tadikonda
Department of Software Engineering and
Game Design, Kennesaw State University
dtadikon@students.kennesaw.edu

***Abstract*—Fault localization is one of the most important activities in software debugging. Among various fault localization techniques, mutation-based fault localization (MBFL) has been commonly studied with its promising performance. However, MBFL should be improved further by incorporating more useful program information. In this paper, we propose MuCatFL, a novel and lightweight technique for better MBFL via mutant categorization. In details, after executing the original test suite against all generated mutants, we categorize the mutants into two groups, positive mutants and negative mutants, to rank the tied program elements. We evaluate MuCatFL by performing an extensive study on 395 real software faults from the widely used benchmark Defects4J. The experimental results show that MuCatFL can significantly outperform MBFL techniques (e.g., localizing 138 faults within the Top-1 position on method-level, 43.75% more than traditional Metallaxis technique). We also investigate that only positive mutants can contribute to the effectiveness of MBFL. Our findings can also provide guidance for the strategies to reduce the execution cost of MBFL.**

***Index Terms*—Software debugging, Fault localization, Mutation testing**

I. Introduction

In modern software development, bugs (a.k.a., faults) are prevalent and inevitable due to the complexity of software systems. They have been widely recognized as notoriously costly and disastrous. For example, Tricentis.com [1] investigated and reported software bugs impacting 3.7 billion users and $1.7 trillion in assets. The first step of debugging is to localize the potential faulty location(s). However, manual fault localization can be time-consuming and error-prone due to the huge code volume. To solve this problem, researchers have proposed various automated fault localization (FL) techniques [2], [3], [4], [5] to help reduce manual efforts. The basic idea of fault localization techniques is to generate a ranked list of program elements (e.g., methods or statements) according to the descending order of their suspiciousness values. Developers can use the ranked list to manually check each element to find and fix the faults in the faulty program. Thus, the target of FL techniques is to rank the faulty elements as high as possible in the ranked list.

In the literature, *spectrum-based fault localization* (SBFL) [6], [2], [7] has been intensively studied since it simply considers the coverage information of failed/passed tests which can be easily collected by many coverage analysis tools. The basic intuition of SBFL is that one program element is more suspicious if it is executed/covered by more failed tests than passed tests. Based on the intuition, various SBFL techniques are proposed (such as Tarantula [2], Ochiai [3], DStar [8], Jaccard [6]) to utilize statistical analysis to compute the suspiciousness values of program elements. Despite the lightweightness, SBFL still has some limitations. For example, some faulty elements may also be covered by passed test cases coincidentally and failed test cases may still cover non-faulty elements. To overcome this limitation, researchers propose *mutation-based fault localization* (MBFL) [5], [4] to consider the actual impact information (a.k.a., killing information) of each program element on the test outcomes by mutating program source code. To localize faulty elements more precisely, typical MBFL techniques such as Metallaxis [4] and MUSE [5] generate mutants for the original program by changing statements syntactically based on predefined rules (also called *mutation operators*, such as changing `a+b` into `a-b`). All the generated mutants are then executed by original test suite and the new execution results of test cases are used for more precise fault localization.

MBFL has been proved to be more effective than SBFL in real bugs [9], but its accuracy is still not promising as expected for many faults. The potential reason is that impact information alone cannot help distinguish some tied program elements (i.e., many elements share same suspiciousness values with the actual buggy elements). Various studies are proposed to improve the accuracy of MBFL. For example, MuSim [10] is presented by identifying the faulty statements based on test case proximity to different mutants. However, these techniques heavily rely on complex computation or analysis (e.g., using test case proximity or neural network). In this paper, therefore, we propose MuCatFL, a novel and lightweight technique for better MBFL by mutant categorization. In details, after executing the original test suite against all generated mutants, we categorize the mutants into two groups, positive mutants and negative mutants, to rank the tied program elements. To evaluate MuCatFL, we perform an extensive study on 395 real software faults from the widely used benchmark Defects4J (V1.2.0) [11]. The experimental results show that MuCatFL can significantly outperform MBFL techniques (e.g.,

DOI:10.18293/SEKE2022-157

localizing 138 faults within the Top-1 position, 43.75% more than traditional Metallaxis technique). This paper makes the following contributions:

- **Technique** A novel and lightweight MBFL technique MuCatFL via mutant categorization.
- **Study** An extensive study on localizing real faults to demonstrate the effectiveness of MuCatFL.

The structure of this paper is as follows. In Section II, we introduce the basis of fault localization and related studies on MBFL. In Section III, we propose the framework and algorithm of MuCatFL. Next, we introduce the experimental study design and analyze the experimental results in Section IV and Section V. Finally, we conclude our paper in Section VI and Section VII.

II. Background and Related Work

A. Fault Localization

Spectrum-Based Fault Localization. The basic idea of SBFL is that program elements covered by more failed test cases tend to be more suspicious. SBFL takes the coverage information between program elements and test suite as input and output a suspicious ranked list in the descending order of suspiciousness values. To date, various SBFL techniques have been proposed such as Tarantula [12], SBI [7], Ochiai [3], Jaccard [6], etc. Even though these techniques use different statistical analysis, they mainly rely on the following components for calculation: (1) the number of all failed/passed tests, i.e., n_f/n_p, (2) the number of failed/passed tests executing program element e, i.e., $n_f(e)$/$n_p(e)$, and (3) the number of failed/passed tests that do not execute program element e, i.e., $n_f(\bar{e})$/$n_p(\bar{e})$. For example, SBI formula can calculate the suspiciousness value of element e as $Susp(e) = \frac{n_f(e)}{n_f(e)+n_p(e)}$. All program elements are ranked based on their suspiciousness values calculated from the formulae and the ranked list is provided to developers to check and repair bugs manually.

Mutation-Based Fault Localization. SBFL has one major limitation. In buggy programs, failed tests may cover non-buggy program elements that do not contribute to the program failure and buggy program elements are also executed by passed tests. MBFL techniques overcome this limitation by considering more effective impact information between test suite and program elements. The first MBFL technique Metallaxis [13], [4] is proposed based on the following intuition: if one mutant impacts failed tests (e.g., the test outcomes change after mutation), its corresponding program element may have caused the failures so that this element should have higher probability to be faulty than others. In Metallaxis, mutants that have impacts on tests are viewed as program elements covered by the tests while the others as uncovered. By simulating the coverage information, Metallaxis applies traditional SBFL formulae to calculate each mutant's suspiciousness value. Finally, the maximum value of mutants is treated as the suspiciousness value of corresponding program element. For example, based on the SBI formula, the suspiciousness value of mutant m can be calculated as $Susp(m) = \frac{n_f^{(m)}(e)}{n_f^{(m)}(e)+n_p^{(m)}(e)}$, where $n_f^{(m)}(e)$/$n_p^{(m)}(e)$ is the number of failed/passed tests whose outcomes are changed due to the mutant m on element e. Another popular MBFL technique is called MUSE [5] which shares similar intuitions with Metallaxis that mutating faulty program elements may cause more failed tests to pass than mutating correct elements and mutating correct elements may cause more passed tests to fail than mutating faulty elements. Besides the two basic MBFL techniques, TraPT [9] extends them to obtain more detailed impact information by transforming test outcomes to extract various test failure messages for better fault localization. For example, TraPT considers MBFL results with the following four different types of test outcomes: (1) Type1: pass/fail information, (2) Type2: exception type, (3) Type3: exception type and exception message, and (4) Type4: exception type, message, and the full stack trace. The four types of test outcomes will generate different impact information. Please note that Metallaxis uses Type4 test failure outcomes and MUSE uses Type1 test failure outcomes according to TraPT.

B. Improvement of Mutation-Based Fault Localization

Previous studies [4], [5], [13] have demonstrated the effectiveness of basic MBFL. However, MBFL still has two major limitations regarding its efficiency and accuracy. The first limitation is that MBFL suffers from the extremely high mutant execution cost problem since it generates a significant number of mutants for the program under test, and each mutant must be executed by all test cases [14], [15]. To overcome this limitation, various studies have been proposed. FTMES [16] is proposed to use only failed tests to execute against mutants and avoid the execution of passed test cases by replacing the impact information with coverage information. IETCR [15] is introduced to reduce the execution of test cases by calculating the entropy change of tests and selecting a proportion of them according to the entropy values. SMBFL [17] is proposed to reduce the execution cost by examining only the statements in the dynamic slice of the program under test to reduce the number of statements to be mutated. Another limitation of MBFL is that impact information alone cannot distinguish many tied program elements so that more advanced program features should be further extracted. For example, MuSim [10] is presented by identifying the faulty statements based on test case proximity to different mutants, 33.21% more effective than existing fault localization techniques such as DStar, Tarantula and Ochiai. In this paper, we propose a novel and lightweight approach to improve the accuracy of MBFL, but our results and findings can provide valuable guidance for more efficient MBFL.

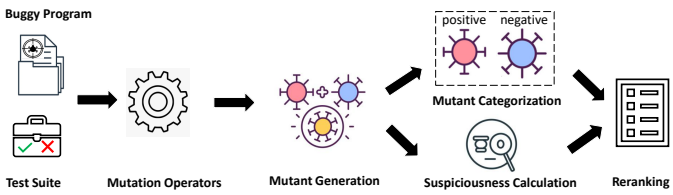

Fig. 1: MuCatFL framework

III. Approach

In this section, we introduce the framework (Section III-A) and algorithm (Section III-B) of our new technique MuCatFL. We also present a real-world example in Section III-C.

Algorithm 1: MuCatFL Algorithm

Input: Faulty program P, test suite T, coverage information C, SBFL formula F, a group of mutators Op
Output: A ranked list S

1 $P' \leftarrow$ A set of elements that covered by failed tests based on the coverage C
2 **for** *element* $e \in P'$ **do**
3 $M(e) \leftarrow$ a set of mutants for e based on Op
4 $N_{positive}(e) \leftarrow 0$ which is the number of positive mutants for e
5 $N_{negative}(e) \leftarrow 0$ which is the number of negative mutants for e
6 **for** $m \in M(e)$ **do**
7 $Sus(m) \leftarrow$ $F(n_p^{(m)}(e), n_f^{(m)}(e), n_f^{(m)}(\bar{e}), n_p^{(m)}(\bar{e}))$
8 **if** $n_f^{(m)}(e) > 0$ **then**
9 $N_{positive}(e)$ ++
10 **else**
11 **if** $n_p^{(m)}(e) > 0$ **then**
12 $N_{negative}(e)$++
13 **end**
14 **end**
15 **end**
16 $Sus(e) \leftarrow$ Max($Sus(m)$)
17 **end**
18 List $S' \leftarrow$ ranked elements according to initial suspiciousness values (descending order)
19 List $S'' \leftarrow$ ranked elements from S' according to the number of positive mutants (descending order)
20 List $S''' \leftarrow$ ranked elements from S'' according to the number of negative mutants (ascending order)
21 Final ranked list S $\leftarrow S'''$

A. Framework of MuCatFL

Figure 1 shows the framework with detailed procedures of MuCatFL. The first several steps are same with traditional MBFL. Given a buggy program under test and its test suite, traditional MBFL techniques apply mutation testing to generate a huge number of mutants for each program statement based on predefined mutation operators (also called mutators). Next, the test suite including all failed and passed test cases is executed against all mutants to record the impact information. The preliminary suspiciousness values can be calculated based on various formulae to get the initial ranked list of the program elements.

In many cases, even the impact information is adopted, some program elements still are tied with the same suspiciousness value. To overcome this limitation, in MuCatFL, we also collect various types of mutants for each program element based on the impact information. We define the following mutant categories based on Type1 failure message (pass/fail information): (1) **positive mutant** that makes **any** failed test pass, (2) **neutral mutant** that makes all test cases unchanged and (3) **negative mutant** that makes **all** failed tests still fail and makes any passed test fail. Since neutral mutants do not contribute to the impact information (no change of test outcomes), we only consider positive mutants and negative mutants in the detailed implementations. Please note that the category definitions based on Type4 failure message is different due to the different impact information (e.g., Type1 for MUSE and Type4 for Metallaxis). For example, the positive mutant according to Type4 failure message represents the mutant that makes the exception type, message and full stack trace of **any** failed test change (even the failed test still fails on the mutant). According to the categories of mutants, we further rank the tied program elements to break the ties based on the following intuitions: (1) if a program element can generate more positive mutants, it has higher probability to be faulty, and (2) if a program element can generate more negative mutants, it has lower probability to be faulty. To this end, for the tied program elements with same suspiciousness value computed by traditional MBFL, we rank the element **higher** if it generates more positive mutants than others. If there are still tied program elements, we further rank the element **lower** if it generates more negative mutants than others. Finally, we can get the final ranked list for all program elements.

B. Algorithm of MuCatFL

Algorithm 1 describes more implementation details of MuCatFL. The entries of MuCatFL include faulty program P, test suite T (with passed tests and failed tests), coverage information C between P and T. It also includes SBFL formula F used by MBFL techniques and a group of predefined mutation operators Op. Next, we explain the details of the algorithm. As the previous study TraPT [9], only suspicious mutants occurring on program elements executed by failed tests contribute to MBFL, so we collect a set of elements P' covered by failed tests in Line 1. From Line 2 to Line 17, we iterate all elements from P' to collect required components for MuCatFL. In Line 3, we generate a set of mutants for each element e based on a group of predefined mutation operators Op. In Line 4 and Line 5, we initialize two variables $N_{positive}(e)$ and $N_{negative}(e)$ to store the numbers of positive mutants and negative mutants for each element. From Line 6 to Line 15, we iterate all mutants for e to calculate their suspiciousness values based on the basic MBFL. We also check if the mutant is positive or negative. In detail, we first check that if $n_f^{(m)}(e)$ is greater than 0 then we add 1 to $N_{positive}(e)$. Otherwise, if $n_p^{(m)}(e)$ is greater than 0, we add 1 to $N_{negative}(e)$. In Line 16, we assign the maximum suspiciousness value of all mutants of e as its final value. Finally, we rank all elements based on the orders of suspiciousness values, the number of positive and negative mutants to get the final ranked list (Line 18 to Line 21).

```
public LegendItemCollection getLegendItems() {
     ...
    int index = this.plot.getIndexOf(this);
    CategoryDataset dataset = this.plot.getDataset(
        index);
--- if (dataset != null) {
+++ if (dataset == null) {
        return result;
    }
    int seriesCount = dataset.getRowCount();
    ...
}
```

Fig. 2: Example of buggy and fixed statements from Chart-1

```
public DefaultDrawingSupplier(Paint[] paintSequence,
     Paint[] fillPaintSequence,...) {

        ...
        this.fillPaintSequence = fillPaintSequence;
        ...
        ...
    }
```

Fig. 3: Example of non-buggy statement from Chart-1

C. Example of MuCatFL

In this section, we use a real-world example from Defects4J (V1.2.0) [11], a widely used Java bug benchmark in the field of software testing and debugging, to demonstrate the effectiveness of our technique MuCatFL. We use Chart-1 which denotes the first buggy version from JFreeChart [18] project. The buggy statement (i.e., `if (dataset != null)` is located in the method `getLegendItems()` of Class `AbstractCategoryItemRenderer` as shown in Figure 2. Based on the traditional MBFL, this buggy statement shares the same suspiciousness value with one non-buggy statement in Class `DefaultDrawingSupplier` as shown in Figure 3. By means of mutant categorization, we observe that there are two positive mutants generated on this buggy statement that can make the failed test pass according to the mutators `RemoveConditionalMutator` and `NegateConditionalsMutator` from the widely used mutation testing tool PIT [19] while there is only one positive mutant generated by the mutator `MemberVariableMutator` for the non-buggy statement. This example demonstrates the effectiveness of MuCatFL to differentiate the buggy elements and non-buggy elements via mutant categorization, indicating that program elements with more positive mutants are prone to be faulty.

IV. Study Design

In this work, we aim to investigate the following research questions:

- **RQ1:** How does MuCatFL perform in localizing real faults compared with traditional MBFL techniques?
- **RQ2:** How do positive mutants or negative mutants impact the performance of MuCatFL separately?
- **RQ3:** What mutators can generate most positive mutants for the studied real-world faults?

A. Implementation and Tool Supports

In this paper, we perform on-the-fly bytecode instrumentation using ASM [20] and Java Agent [21] to collect the required coverage information. We apply the widely used mutation testing framework PIT (Version 1.1.5) [19] to perform mutation testing. Following the previous study TraPT [9], we use all 16 mutation operators available in PIT-1.1.5 and modify PIT to collect the required impact information. For example, we modify PIT to enable it executing on programs with failed tests and force it to execute each mutant against the remaining tests even the mutant is killed by earlier tests. We implement MUSE and 5 widely used traditional SBFL formulae (Tarantula [2], Ochiai [3], DStar [8], Jaccard [6] and SBI [7]) for Metallaxis. We use 395 faulty versions from all the 6 projects (Lang, Chart, Time, Math, Mockito and Closure) in widely used Defects4J benchmark (V1.2.0) [11].

B. Evaluation Metrics

Many prior studies [22], [23], [24], [25] perform fault localization techniques on method-level, i.e., localizing faulty methods among all source code methods, since statement-level fault localization may be too fine-grained without context information [26] and class-level fault localization is too coarse-grained [27]. In these studies, the suspiciousness value of one method is assigned as the the maximum suspiciousness value of all mutants generated in this method. In this paper, we also evaluate MuCatFL on method-level inspired by other studies but make some changes since the number of mutants is involved. In detail, we firstly rank all statements in each source code method based on Algorithm 1. Next, we find the top-rank statement and assign its suspiciousness value, the number of positive mutants and the number of negative mutants to its corresponding method. Finally, we rank all source code methods based on the Line 18-20 in Algorithm 1. We use following evaluation metrics to evaluate various MBFL techniques. (1) Top-N (Top-1, Top-3, and Top-5 in our study) metric simply represents the exact position of the buggy elements in the ranked list. The motivation to use Top-N metric is that most developers will stop using debugging tools if they cannot return the actual buggy elements within the Top-5 positions [27]. (2) MFR (mean first rank). For a buggy version with multiple buggy elements, we use MFR to compute the mean of the first buggy element's rank for each buggy version since the localization of the first buggy element can be a guide to the rest of buggy elements. (3) MAR (mean average rank) is simply the mean of the average of all buggy elements' ranks for each buggy version.

V. Result Analysis

A. RQ1 - Performance of MuCatFL

In this section, we investigate the effectiveness of MuCatFL compared with traditional MBFL techniques (Metallaxis and MUSE). Figure 4 shows the overall fault localization results on all studied subjects (i.e., Lang, Chart, Time, Math, Mockito and Closure from the Defects4J benchmark) in terms

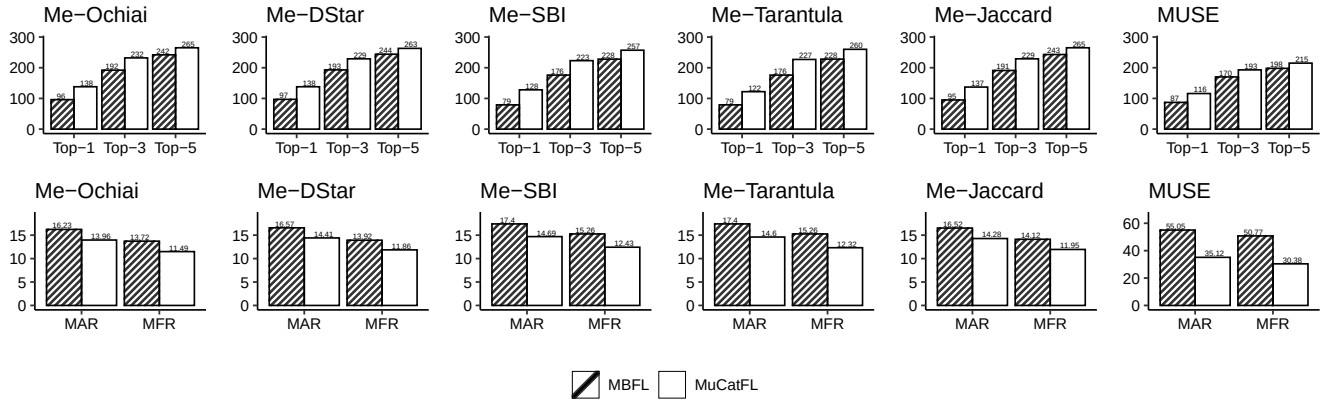

Fig. 4: Results of MuCatFL compared with Metallaxis and MUSE

TABLE I: Impacts of positive and negative mutants

Tech Name	Top-1	Top-3	Top-5	MFR	MAR
Me-Ochiai	96	192	242	13.72	16.23
MuCatFL(P and N)	138	232	265	11.49	13.96
MuCatFL(Only P)	133	230	263	12.08	14.71
MuCatFL(Only N)	104	205	246	12.93	15.28

of Top-1, Top-3, Top-5, MFR and MAR. The upper sub-figures represent the Top-N results and bottom sub-figures indicate the MFR/MAR results. Each pair of bars in the sub-figures represents the comparison between MuCatFL and one traditional MBFL with different formulae. Please note that in the figure we use "Me-formula" to represent Metallaxis with corresponding SBFL formula. In these figures, higher Top-N value and lower MFR/MAR value indicate a better localization technique. From the figures, we have following observations. First, MuCatFL with mutant categorization outperforms traditional MBFL techniques for **all** SBFL formulae. For example, in total, Metallaxis with Ochiai formula can localize 96 faulty methods within Top-1, while MuCatFL is able to localize 138 faulty methods, 43.75% more effective than traditional MBFL technique. Furthermore, in terms of MAR, MuCatFL for Ochiai is 13.96, 13.99% more precise than Metallaxis with Ochiai (16.23). Second, in terms of MAR/MFR, MuCatFL can improve Metallaxis by less than 20% for the five formulae. However, MUSE can be improved by 36.2% and 40.16% when we consider different categories of mutants. The potential reason can be that MUSE only considers pass/fail information so that there are more rooms to be improved by utilizing positive mutants and negative mutants.

B. RQ2 - Impacts of Positive Mutants or Negative Mutants

In the RQ1, we compare MuCatFL with traditional MBFL techniques by considering both positive and negative mutants. However, whether both of them contributes to MuCatFL has not been studied. In this section, we investigate the effectiveness of MuCatFL by considering positive or negative mutants **separately**. Table I shows the fault localization results with only Ochiai formula for different configurations since MuCatFL with Ochiai can achieve the best performance according to RQ1. In this table, Me-Ochiai represents traditional MBFL technique and MuCatFL (P and N) indicates MuCatFL with both positive and negative mutants. Also, MuCatFL (Only P) represents MuCatFL only considering positive mutants while MuCatFL (Only N) represents MuCatFL only considering negative mutants. From the table, we have following observations. First, both positive and negative mutants are helpful for MuCatFL. For example, in terms of Top-1, MuCatFL (P and N) can localize 138 faulty methods within Top-1, more than any other configurations. Second, only positive mutants can still contribute to promising performance of MuCatFL. In detail, MuCatFL (Only P) can help localize 133 faulty methods within Top-1, very close to MuCatFL (P and N). However, MuCatFL with only negative mutants performs worse than that with only positive mutant (localizing 104 bugs within Top-1). Such findings demonstrate that failed tests should be more important than passed tests when localizing faults for MBFL, indicating the potential improvement of accuracy for some cost reduction strategies (e.g., FTMES [16] with the execution of only failed tests against all mutants).

C. RQ3 - Impacts of Mutation Operators for Positive Mutants

In MBFL, we generate mutants by applying different mutation operators, and the findings in RQ2 show that only positive mutants can contribute to MuCatFL. In this section, we investigate what mutation operators can mostly lead to positive mutants in terms of both Type1 and Type4 test failure messages. We count the number of positive mutants with their corresponding mutators in Table II and Table III, accordingly. Please note that we only include 6 most frequent mutators in the tables, which can reveal some interesting findings. In the two tables, the first column represents the mutators from PIT and the second column indicates the number of positive mutants generated by the corresponding mutators. From the two tables, we can find that the top 6 mutators are exactly same for Type1 and Type4 failure message even the orders are different, indicating that program elements that can be mutated by these mutators tend to be faulty. This finding can be also applied to reduce the huge execution cost of MBFL. For example, mutants generated by top frequent mutators can have higher execution priorities, or only the top frequent mutators can be selected to generate mutants.

TABLE II: Mutators generating most positive mutants in terms of Type1 failure message

Mutator	# of positive mutants
NonVoidMethodCallMutator	4686
NegateConditionalsMutator	4170
RemoveConditionalMutator_EQUAL_ELSE	2698
InlineConstantMutator	1998
ReturnValsMutator	1960
RemoveConditionalMutator_EQUAL_IF	1912

TABLE III: Mutators generating most positive mutants in terms of Type4 failure message

Mutator	# of positive mutants
NonVoidMethodCallMutator	115912
NegateConditionalsMutator	66043
ReturnValsMutator	47439
InlineConstantMutator	47218
RemoveConditionalMutator_EQUAL_IF	45171
RemoveConditionalMutator_EQUAL_ELSE	31347

VI. Threats to Validity

The main threat to *internal* validity is from our implementation. To reduce this threat, we implement our techniques by utilizing state-of-the-art tools and frameworks, such as ASM and PIT. The main threat to *external* validity mainly lies in the selection of the studied subjects. To reduce this threat, we evaluate on more real-world projects. The main threat to *construct* validity is that the measurements used may not fully reflect real-world situations. To reduce this threat, we use Top-N, MAR and MFR metrics, which have been widely used in previous studies [9], [25], [28], [24].

VII. Conclusion

In this paper, we propose MuCatFL, a novel and lightweight technique for better MBFL via mutant categorization. In details, after executing the original test suite against all generated mutants, we categorize the mutants into two groups, positive mutants and negative mutants, to rank the tied program elements. We evaluate MuCatFL by performing an extensive study on 395 real software faults from the widely used benchmark Defects4J. The experimental results show that MuCatFL can significantly outperform MBFL techniques (e.g., localizing 138 faults within the Top-1 position, 43.75% more than traditional Metallaxis technique).

References

[1] "Tricentis reports," 2018. [Online]. Available: https://www.tricentis.com/blog/how-to-avoid-the-tricentis-software-fail-watch/

[2] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 273–282.

[3] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 2006, pp. 39–46.

[4] M. Papadakis and Y. Le Traon, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[5] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 2014, pp. 153–162.

[6] 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-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[7] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *ACM Sigplan Notices*, vol. 40, no. 6, pp. 15–26, 2005.

[8] W. E. Wong, V. Debroy, Y. Li, and R. Gao, "Software fault localization using dstar (d*)," in *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*. IEEE, 2012, pp. 21–30.

[9] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133916

[10] A. Dutta, A. Jha, and R. Mall, "Musim: Mutation-based fault localization using test case proximity," *International Journal of Software Engineering and Knowledge Engineering*, vol. 31, no. 05, pp. 725–744, 2021.

[11] 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 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

[12] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization for fault localization," in *in Proceedings of ICSE 2001 Workshop on Software Visualization*, 2001.

[13] M. Papadakis and Y. Le Traon, "Using mutants to locate" unknown" faults," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 691–700.

[14] M. Kooli, F. Kaddachi, G. Di Natale, A. Bosio, P. Benoit, and L. Torres, "Computing reliability: On the differences between software testing and software fault injection techniques," *Microprocessors and Microsystems*, vol. 50, pp. 102–112, 2017.

[15] H. Wang, B. Du, J. He, Y. Liu, and X. Chen, "Ietcr: An information entropy based test case reduction strategy for mutation-based fault localization," *IEEE Access*, vol. 8, pp. 124 297–124 310, 2020.

[16] A. A. L. de Oliveira, C. G. Camilo-Junior, E. N. de Andrade Freitas, and A. M. R. Vincenzi, "Ftmes: A failed-test-oriented mutant execution strategy for mutation-based fault localization," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2018, pp. 155–165.

[17] N. Bayati Chaleshtari and S. Parsa, "Smbfl: slice-based cost reduction of mutation-based fault localization," *Empirical Software Engineering*, vol. 25, no. 5, pp. 4282–4314, 2020.

[18] "Jfreechart website," 2022. [Online]. Available: https://github.com/jfree/jfreechart

[19] "Pit mutation testing system," 2022. [Online]. Available: http://pitest.org/

[20] "Asm java bytecode manipulation and analysis framework," 2022. [Online]. Available: https://asm.ow2.io/

[21] "Java programming language agents," 2022. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html

[22] T. Dao, L. Zhang, and N. Meng, "How does execution information help with information-retrieval based bug localization?" in *Proceedings of the 25th International Conference on Program Comprehension*, 2017, pp. 241–250.

[23] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 169–180.

[24] M. Zhang, X. Li, L. Zhang, and S. Khurshid, "Boosting spectrum-based fault localization using pagerank," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 261–272.

[25] J. Sohn and S. Yoo, "Fluccs: using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 273–283.

[26] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 international symposium on software testing and analysis*, 2011, pp. 199–209.

[27] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 165–176.

[28] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 191–200.