On the relationship of repair actions and patterns on bug localization approaches: a comparative study

Júlia Manfrin Dias*
julia.dias@ufu.br
Universidade Federal de Uberlândia
Uberlândia, MG, Brazil

Marcelo de Almeida Maia*
marcelo.maia@ufu.br
Universidade Federal de Uberlândia
Uberlândia, MG, Brazil

ABSTRACT

In the evolving field of automatic program repair, bug localization approaches play an important role, given their potential to significantly narrow down the search range for potential solutions. Given the diversity of bug locating approaches and their respective performance variations, this study aims to evaluate the bug localization accuracy, considering specific characteristics of the bugs, defined by actions and repair patterns observed in patches. Repair actions constitute changes made to the code to derive a suitable solution to a given problem. These actions can range from additions, deletions, and alterations in the source code lines. Repair patterns, on the other hand, are generalized representations of recurrent action structures in the fixed code.

We evaluate the accuracy of each bug locator with respect to these actions and repair pattern to identify if there are any influence of specific actions and/or patterns in each type of locator.

KEYWORDS

bug localization, action patterns, repair patterns, Defects4J

1 INTRODUCTION

Bug localization approaches play a crucial role in automatic program repair. They are responsible for identifying the locations in the code where bugs exist, which is a critical initial step in the automatic program repair process. Proper bug localization reduces the search space of potential fixes, making the repair process more efficient. Without effective bug localization, an automatic repair tool might end up scanning the entire codebase, wasting resources and time, and potentially proposing fixes in wrong locations.

Bug localization is a step in the code correction process that seeks to identify the class, method, or even code elements causing system failures. The whole process is time-consuming for locating and fixing bugs, subsequently leading to a decrease in productivity and efficiency for the team developing the project. With the increased software demand over the years, identifying and fixing errors in the codes has become costly and exhaustive for developers and companies. In response, many researchers have dedicated to studying and proposing techniques to automate this bug localization phase [11].

Bug localization approaches, aside from potentially easing the manual work of developers to repair bugs, are directly important to automatic bug repair tools, such as GenProg[5], Nopol[12], and NPEFix[1]. This importance arises from the fact that bug localization is a step in automatic bug repair tools, which generally have a stage for bug identification, bug localization, and finally, synthesis and application of repair.

However, recent literature on automatic software repair has shown that there are bug characteristics that influence the performance of software repair tools regarding their accuracy. Therefore, several benchmarks for evaluating bug repair tools have been proposed with the aim of checking how well a tool performs effectively on bugs of different nature[3, 4, 7].

This study aims at indirectly contributing to the improvement of automated bug localization systems by understanding if the actions and patterns found in bug repairs can influence the performance of different approaches. More specifically, this work aims to verify if there is some relationship between the performance of different bug localization approaches and the type of repair action/pattern of bugs. To organize the study, we pose the following research questions:

- RQ1: How do the locators perform on bugs with distinct repair actions according removal/modification/addition?
- RQ2: How do the locators perform on bugs with distinct repair actions according the affected syntactic structure?
- RQ3: How do the locators perform on bugs with distinct repair patterns?

In the following, we present a review of bug localization approaches and the considered action and repair patterns. In Section 3, we present the study setting. In Section 4, we report the found results. Then, we present a brief section on related work and finally the conclusion.

2 BACKGROUND

2.1 Bug Localization Approaches

2.1.1 Ochiai and DStar. The Ochiai and Dstar are fault localization approaches based on the failing and successful test cases. The more a code element or command passes the failing test cases and the less it passes the successful ones, the more suspect that element is to contain the error. The difference between the Dstar approach and Ochiai is the mathematical formula used to rank the suspicious code elements.

Ochiai formula:

$$S(s) = \frac{failed(s)}{\sqrt{totalfailed \cdot (failed(s) + passed(s))}} \quad (1)$$

DStar formula:

$$S(s) = \frac{failed(s)^*}{passed(s) + (totalfailed - failed(s))} \quad (2)$$

Where:

- s is a code instruction.
- *S(s)* is the Score.

- *failed(s)* are the test cases in which the instruction *s* failed.
- *passed(s)* are the test cases in which the instruction *s* passed.
- *totalfailed* are all the test cases in which the program failed.
- In the DStar formula, the * is a variable exponent. The article by [9] considers this value to be 2.

2.1.2 Metallaxis and Muse. The Metallaxis and Muse approaches use the concept of mutation to discover errors in the code. Mutation means swapping one code element, which can be an operator or a command, for another element [8]. After a mutation is performed, the code is submitted to the test cases, following the same steps as the Ochiai and Dstar techniques. The more the code passes the failing test cases and the less it passes the successful ones, the more suspected of containing the bug that code is considered.
Both Metallaxis and Muse use formulas to calculate the suspects and generate a ranked list of suspects. The difference between Metallaxis and Muse is also in the formula, as shown below:
Metallaxis formula:

$$S(m) = \frac{failed(m)}{\sqrt{totalfailed \cdot (failed(m) + passed(m)}} \quad (3)$$

Muse formula:

$$S(m) = failed(m) - \frac{f2p}{p2f} \cdot passed(m) \quad (4)$$

Where:
- *m* is the command that underwent the mutation.
- *S(m)* is the Score.
- *failed(m)* are the test cases where the command *m* failed.
- *passed(m)* are the cases where the command *m* passed.
- *f2p* (failed to passed) are the numbers of test cases where the command *m* failed and after the mutation it passes.
- *p2f* (passed to failed) are the numbers of test cases where the command *m* passed and after the mutation it fails.

2.1.3 Slicing with Union, Slicing with Frequency, and Slicing with Intersection. Slicing-type approaches use the concept of creating a set with the lines of the program (commands) that can interfere with the value of a variable. This set is considered the slicing of the program [13].
When a bug appears in the program, usually some variable of this program presents an incorrect value and the program is paused. It is from this incorrect variable that the algorithm slices the program.
The slicing occurs with the identification of all the sentences in the code, from the beginning of the program, that directly or indirectly affects that variable with an undesirable value. After this identification process, a set (the slice) is created with all those sentences that indeed contribute to a variable's value coming out as expected or not.
This approach uses the failed test cases to identify the variables with wrong values and apply the slicing algorithm. As there can be several failed tests, more than one slice is created. Some strategies have emerged within this slicing approach that uses the different slices generated, which are:
- Slicing: This approach uses the union of the slices generated by the algorithm to generate the list of suspect elements. The diagram below exemplifies the union of the slices in slicing union.
- Slicing_count: This strategy checks how many times a sentence, command or element is included in the code slice. The more frequent the element is, the more suspicious.
- Slicing_intersection: It uses the intersection of several code slices to calculate the elements suspected of errors. The slicing intersection algorithm only considers the sentences that are common in the slices created.

2.1.4 Stacktrace. As a bug localization approach, Stacktrace uses this stack to find bugs and thus suggest instructions that contain defects. The algorithm starts with the execution stack stacking each method call that is executed. Each of these calls is allocated in a stack frame. When an exception occurs, the program and stack are paused. With this, it is observed in the stack the methods called and the last method executed before the failure, which is in the first stack frame, is considered the probable cause of the problem.

2.1.5 Predicate Switching. The Predicate Switching approach is a bug localization technique focused on predicates and program flow controls. A predicate is defined as a sentence that assumes a logical value (true or false). Flow controls are conditional structures that can make different decisions depending on their input values. Thus, a predicate can control the execution of different branches in the system [13]. Tthe algorithm traverses the execution stack, from a failed test, and identifies all branches. Then the test is run several times, but the predicate that generated the branch undergoes a mutation, so as to generate a different result each time.
Thus, the algorithm executes the failed test case multiple times, applying mutations to the predicate until one of them leads to a branch that passes the test case. If this modification in the predicate produced the correct output, then this predicate is called a critical predicate [13] Predicate Switching differs from the approaches that use mutation in terms of mutations in the control flow, not in the code.

2.2 Repair Actions and Patterns

After a bug is located in the code, it is repaired so that the program in question operates correctly. The repair that occurs in the bug can be done in several ways. New lines may be added to the code, or some elements or even lines may be removed. Alternatively, some elements may just be modified without the need to add or remove lines.
It was from these events or actions that occur to fix the bug, that Sobreira and colleagues [10] observed that they could categorize these actions that take place for the repair to occur. Thus, repair actions were defined as blocks of source code modified/inserted/deleted to make repairs. Repair patterns are higher-level abstractions made from actions.

2.2.1 Repair Actions. These are the simplest blocks to make a repair. They encompass actions of additions of lines or commands, removal of lines or commands, and modification of commands. In the article by [10], 68 actions were identified and categorized using Defects4j bugs.

2.2.2 Repair Patterns. Patterns are high-level abstractions observed from actions. While repair actions define simpler actions made in the code, such as a line removal, repair patterns are defined as those actions that occur with a certain frequency in bug repairs, resembling patterns.
For example, the addition of an 'if' is a repair action of the addition type, and a 'for' that was removed would also fit into the same action. When this action occurs in other bugs, it can be determined that a repair pattern has been found, defined as a 'Conditional Block' type pattern. Nine repair patterns were identified [10].

3 STUDY PROPOSAL

The proposal of this work is to present an empirical study on the relationship between bug localization approaches and the properties of bugs, more precisely the actions and repair patterns in order to evaluate the effectiveness of LB's in terms of bugs fixed.
The accuracy data used to investigate the research hypothesis include the results extracted from the studies[1] by [13] and the used bug dataset is the Defects4j benchmark [3]. The collected data contains information about how bug localizers scores the locations of the target system and marking the as faulty or not. Information about the bugs from Defects4J was gathered from the work of Sobreira and colleagues[2][10]. With this information, it was possible to cross-reference which actions and repair patterns were contained in the bugs. To answer the research questions the data was organized into tables to visualize and process the data.
Table 1 represents which localizer approaches managed to find the error in the respective bug, where number 1 indicates that the locator in that column actually found the bug failure and 0 when the locator did not find the bug error.
Table 2 shows which actions and repair patterns exist in the bugs. The number 1 indicates that the bug has that action or repair pattern. A bug can have more than one action and pattern. The number 0 indicates that the bug does not have that action/pattern.
The Table 3 indicates the sum of bugs that have a particular action that the localizer managed to find within the 357 bugs. For example, out of the 357 bugs, the localizer 'Ochiai' was able to find the error in 30 bugs that require action 'assignAdd' for repair, and so forth.

4 RESULTS

In this section, the results obtained to answer the research questions are presented.

4.1 RQ1: How do the locators perform on bugs with distinct repair actions according removal/modification/addition?

The first analysis focused on separating the repair actions only into 3 groups. The groups were as follows: the Removal

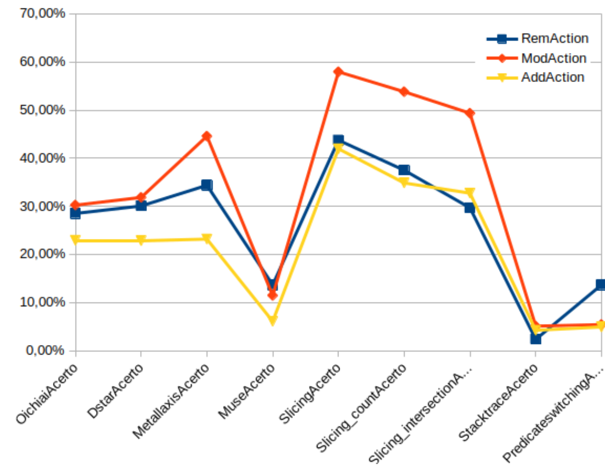

Figure 1: Localizers' success rate for actions divided into 3 groups

group, which includes the removal of commands or lines; the Modification group, which includes actions that make some modification to commands or a modification of an element in a line of code; and the Addition group, which includes actions that add lines to the code.
The graph below shows the 3 groups of actions and the success rate of the localizers in finding the bug with those actions.
Some observations can be made:

- The bug localizers had better performance in terms of the number of bugs found with the bug group with Modification type actions.
- The best performing locators in terms of the number of bugs found were those with the Slicing approach, reaching a success rate to locate errors of 60
- Only the Muse and PredicateSwitching locators managed to find better performance in terms of bugs found for the bug group with Remotion action, despite the rates being lower.

4.2 RQ2: How do the locators perform on bugs with distinct repair actions according the affected syntactic structure?

The second analysis divided the actions into 10 groups, following the classification in [10]. In this grouping, there was no distinction between actions that add, remove or modify lines of code. The formed groups were as follows, accordingly to the syntactic structures where the actions have taken place: 1) Assignment; 2) Conditional blocks; 3) Loop blocks; 4) Method Call; 5) Method Definition: actions that define a method; 6) Object Instantiation 7) Exception handling and throwing; 8) Return command; 9) Variable declarations; 10) Type, which are actions that involve data Types, such as int, float, double, boolean, String.

[1]https://damingz.github.io/combinefl/index.html
[2]https://program-repair.org/defects4j-dissection

Project	Bug	Faulty	Oichiai	Dstar	Metal.	Muse	Slicing	Sl._count	Sl._inters.	St. trace	Pred. Sw.
Chart	1	1	0	0	0	0	1	1	1	0	0
Chart	2	1	0	0	0	0	1	0	0	0	0
...											
Time	25	1	0	0	0	0	0	0	0	0	0
Time	27	1	0	0	0	0	1	1	1	0	0

Table 1: Hits of Bugs vs Locators

Project	BugId	assignAdd	assignRem	assignExpChange	. . .	codeMove
Chart	1	0	0	0	. . .	0
Chart	2	1	0	0	. . .	0
...						
Time	27	0	0	0	. . .	0

Table 2: Bugs vs Actions and Repair Patterns

Localizers	assignAdd	assignRem	assignExpChange	condBranIfAdd	. . .	notClassified
Oichiai	30	10	18	31	. . .	4
Dstar	31	11	20	30	. . .	4
Metallaxis	35	11	28	32	. . .	6
Muse	10	5	5	9	. . .	1
Slicing	67	16	34	63	. . .	5
Slicing_count	55	14	32	52	. . .	5
Slicing_intersection	52	12	29	46	. . .	5
Stacktrace	5	0	3	7	. . .	0
Predicateswitching	7	6	0	7	. . .	0

Table 3: Total of bugs with actions and repair patterns by locator

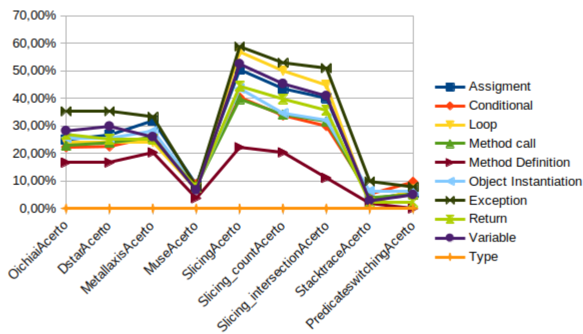

Figure 2: Localizers' success rate for actions divided into 10 groups

The graph below shows the success rate of locators with respect to the number of bugs found by the actions divided into these 10 groups of actions needed to fix the bug.
From this graph, some observations can be made:

- The graph has a behavior quite similar to the graph of actions divided into 3 groups, as the localizers of the Slicing approaches also proved to be better in this experiment.
- The Exception action, which are actions that involve the handling and throwing of exceptions, was the most accurate on the all locators.
- The Method Definition and Type actions underperforms on all locators.

4.3 RQ3: How do the locators perform on bugs with distinct repair patterns?

The third analysis considered the performance of bug localizers, in terms of the number of bugs found, in relation to the patterns defined in [10]. The repair patterns are as follows:

(1) WrongVarRef: variables were referenced incorrectly.
(2) WrongMethodRef: a method is referenced incorrectly.
(3) MissNullCheckP: The 'P' indicates positive null check, an example of this command is: 'x == null'.
(4) MissNullCheckN: The 'N' indicates negative null check, an example of this command is: 'x != null'.
(5) SingleLine: repairs with the addition of a line, or removal or modification of a single line.
(6) CopyPaste: repeat the same modifications at different points in the source code.
(7) ConstChange: make a constant change.

(8) CodeMove: movement of lines of code, i.e., take a section of code and place it elsewhere in the code.
(9) NotClassified: repairs whose actions do not fit into any pattern.

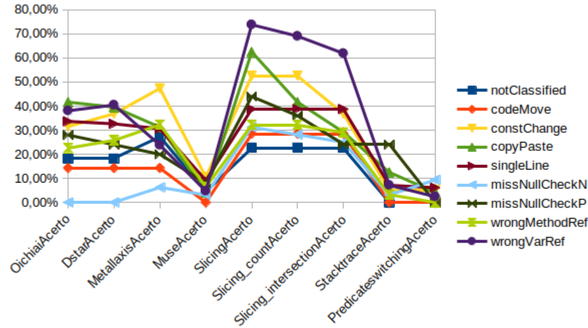

Figure 3: Success rate of locators by repair patterns

Evaluating the result of the graph above, we have that:

- The Slicing approach localizers also proved more efficient in this experiment, in terms of the number of bugs located.
- The Slicing approach achieves a success rate in locating bugs above 80% with the WrongVarRef repair pattern. Other locators also tend to have better performance when this pattern appears.
- CodeMove and Negative Missing Null Check underperforms on all locators.

5 THREATS TO VALIDITY

There is an external threat to validity related to scope of the study. Our study used the Defects4J benchmark, which may not reflect the real distribution of the universe of bugs. Moreover, our study is limited to Java bugs, and may not apply to bugs in other languages.

Concerning, the internal validity, we relied on third-party datasets, that may contain errors regarding the localization of the bugs. Moreover, the actions and patterns of the third-party dissection of the Defects4J would have different classifications if conducted with different types of actions and patterns.

6 RELATED WORK

Other attempts have been made to understand the impact of the bug features on the performance of automated program repair and bug localization approaches. Zou and colleagues observed that different bug localization approach perform better on specific classes of bugs and that combining them improve the overall performance of locators [13]. Our work further explain why the combination of different types of approaches may improve the overall accuracy.

Kui and colleagues already showed that "you cannot fix what you cannot find" and that only a part of Defects4J bugs can be properly located [6]. Our work have shed light on which classes of bugs, regarding the repair actions/patterns, the different locators perform better.

Durieux and colleagues found that automated repair tools tend to overfit specific benchmark as Defects4J [2], indicating that approaches may target on the most prevalent type of bugs in the respective datasets. Our work focus on evaluating only the bug localization part.

7 CONCLUDING REMARKS

Our work has shown that locators work better on bugs that require line modification. Bugs that either require removing or adding lines tend to be more difficult to localize.

Regarding repair actions, we observed that bugs with actions that involve the handling and throwing of exceptions was the most accurate on the all locators. Bugs with repair actions involving types and method definition have poor performance and should be better investigated.

We found that bugs fixed with the Wrong Var Ref pattern were easier to locate. On the other hand, bugs that contain the CodeMove and Negative Missing Null Check underperforms on all locators, and should be better investigated.

As future work, we expect to use the found observations to propose techniques that may improve the bug localization approaches on bugs that have the properties shown above.

Acknowledgments: We acknowledge CAPES, FAPEMIG and CNPq for partial support of this work.

REFERENCES

[1] Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. 2017. Dynamic Patch Generation for Null Pointer Exceptions Using Metaprogramming. In *Proc. of SANER'2017*. 349–358.
[2] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts. In *Proc. of ESEC/FSE'2019)*.
[3] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proc. of ISSTA '14*. ACM, New York, NY, USA, 437–440.
[4] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering* 41, 12 (Dec. 2015), 1236–1256.
[5] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. on Software Engineering* 38, 1 (2012), 54–72.
[6] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. In *Proc. of ICST'2019*. 102–113.
[7] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *Proc. of SANER '2019*.
[8] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: Mutation-Based Fault Localization. *Softw. Test. Verif. Reliab.* 25, 5–7 (aug 2015), 605–628.
[9] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization. In *Proc. of ICSE'2017)*. 609–620.
[10] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. 2018. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In *Proc. of SANER'2018*. 130–140.
[11] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.
[12] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clément, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Tran. on Software Engineering* 43, 1 (April 2016), 34–55.
[13] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D. Ernst, and Lu Zhang. 2021. An Empirical Study of Fault Localization Families and Their Combinations. *IEEE Trans. Softw. Eng.* 47, 2 (feb 2021), 332–347.