Hey APR! Integrate Our Fault Localization Skill: Toward Better Automated Program Repair

Kyosuke Yamate*, Masanari Kondo*, Yutaro Kashiwa*, Yasutaka Kamei* and Naoyasu Ubayashi*
*Kyushu University, Japan
Email: yamate@posl.ait.kyushu-u.ac.jp, (kondo, kashiwa, kamei, ubayashi)@ait.kyushu-u.ac.jp

***Abstract*—*Background:* Prior studies lack the perspective of using developer's skills to augment the performance of *automated program repair (APR)*. APR has a phase referred to as *fault localization (FL)*, which automatically finds the faulty statement that causes faults. To achieve a well-performed FL phase, we study developers' FL skills, which allow developers to find faulty statements. We suppose that such FL skills can add additional information to fault localization to augment the accuracy of fault localization and reduce the execution cost of APR.**
***Aims:* We aim at revealing a criterion that distinguishes whether using the FL skill reduces the execution cost of the state-of-the-art APR, TBar, depending on the accuracy of the FL skill.**
***Method:* We conduct a simulation case study in the Defects4J dataset, which is the most popular dataset. We compare the numbers of candidate patches generated by TBar using the FL skill or using *spectrum-based fault localization (SBFL)*.**
***Results:* Our case study revealed that, if developers localized the faulty statements before inspecting 40% of the statements in the target program, the execution cost of TBar reduces for over half of the studied faults. The 40% value is a requirement for developers using the FL skill to augment the performance of APR.**
***Conclusion:* If developers can localize the faulty statement before inspecting 40% of the statements, integrating the FL skill with SBFL makes TBar faster compared to when SBFL is used.**

***Index Terms*—Automated Program Repair, Fault Localization, Developers**

I. Introduction

Debugging is one of the most time-consuming processes in software development where developers find and fix the faulty statement by their skills based on their experience, knowledge, and instincts. The debugging process accounts for more than 50% of software development costs [1], [2]. To facilitate this process, *Automated Program Repair (APR)* has become an important research field in software engineering [2].

APR consists of two phases: "*fault localization*" and "*patch generation*". The first phase identifies the *faulty statement* that causes faults; the second phase creates patches and validates them to see if each one passes the test case. In particular, the *patch generation* phase heavily relies on the fault localization phase. Even if fault localization specifies wrong statements of code as faulty statements [3], the patch generation phase still generates numerous patches and exercise test cases. As APR repeats these two phases until a generated patch passes the associated test, the process can take a long time, which is the execution cost.

It is, therefore, necessary to study approaches to augment the performance of fault localization to reduce the execution cost of APR. Prior studies often investigate fully automated fault localization. However, no studies exist that intend to reduce the execution cost of APR by using developers' *fault localization skills (FL skills)* to localize faults. Developers' FL skills represent the ability to localize the faulty statement. We suppose that such FL skills can add additional information to fault localization and augment the performance of fault localization to reduce the execution cost of APR. We refer to using developers' FL skills for the fault localization phase in APR as *manual fault localization (MFL)*. However, it is unclear to what extent the FL skills are required to reduce the execution cost of APR.

In this paper, we conducted a simulation case study to clarify to what extent the FL skills are required to reduce the execution cost of APR. When developers intend to find the faulty statement, they often determine the suspiciousness of fault for statements and inspect them from the most suspicious statement. Hence, we use the skill to determine the suspiciousness as the studied developers' FL skill and use it in the fault localization phase in APR. We quantified this skill as the number of inspected statements before developers localized the faulty statement. For the simulation, we set the following central question:

> *How many statements are developers allowed to inspect to reduce the execution cost of APR when using the FL skill?*

To address our central question, we investigated MFL in one of the current state-of-the-art APRs: TBar [4]. This is because TBar provides record performance in the proposed APR methods on Java. This simulation case study employs the Defects4J [5] dataset, which is the most popular dataset in the APR research.

The answer to the central question provides developers with a criterion for deciding whether to use MFL for reducing the execution cost of TBar, depending on the accuracy of their MFL. We compared the number of candidate patches by TBar with MFL with those of TBar using *spectrum-based fault localization (SBFL)* [6], [7]. SBFL is one of the most popular automated fault localization techniques.

The main findings of this study are as follows:

- If developers localized the faulty statement before inspecting 40% of the statements in the Java method, the execution cost of TBar reduces compared to TBar with SBFL over half of the studied faults.

- We found that the number of generated patches varied greatly depending on the inspected statement. This is a side effect of APR, which varies the execution cost. SBFL does not remedy this side effect because it is a fully automated method. On the contrary, MFL might remedy this side effect because developers can avoid selecting the types of statements for which APR generates more patches.

This study's major contributions are as follows: (1) Studying the feasibility of TBar with low execution costs through interactive intervention by developers, (2) Opening a new research perspective in which we consider the interaction between the automated method and developers to augment the capability of APR.

The remainder of this paper is organized as follows. Section II introduces the background of APR. Section III explains our main idea, MFL. Section IV presents the case study setup of our experiment. Section V presents the results of our case study. Section VI discusses the case study. Section VII describes the threats to the validity. Section VIII presents the conclusion and prospects.

II. Background of APR

Prior studies [8], [9], [4], [10], [11], [12] have evaluated various APR methods so far. One of the most popular families of APR methods is *template-based APR* [2], [4]. Template-based APR uses *pre-defined templates* [2], which are a set of change operators for programs such as modifying program conditions, adding code, and removing code. The APR family modifies programs according to the pre-defined templates. These APR methods generate patches to modify programs. The patches are verified by the test cases by the successful modification of the programs or not. However, even if a generated patch can pass all the test cases, it might break a necessary behavior [13]. These patches are often called *plausible patches*. We need to investigate to check whether they are *correct patches* that do not break a necessary behavior.

Before generating patches, APR should localize the faulty statement. In the fault localization phase, APR identifies the faulty statement in programs. SBFL [6], [7] is one of the most popular fault localization techniques used in APR. SBFL detects faulty statements by assigning a suspiciousness score to each statement based on test results and execution paths. The suspiciousness score is a number that indicates the likelihood of the cause of fault for each statement. Generally, statements executed by failed tests have higher suspiciousness scores, whereas those executed by passed tests have lower suspiciousness scores. To compute the suspiciousness score, various calculation methods have been proposed [14], [15].

III. Manual Fault Localization (MFL)

Prior studies [16], [4] showed that the accuracy of fault localization is important for APR. However, prior studies only focused on fully automated FL techniques. To improve the accuracy, we focused on developers' FL skill. Developers often find faulty statements by using the breakpoint of IDE tools and the print function in programming languages (*human debugging*). In the process of human debugging, developers use their FL skills to find faulty statements. Inferring the suspiciousness of fault for statements using the debugging experience is an FL skill. Expert developers may find faulty statements accurately by human debugging using their FL skills. Inspired by human debugging, we study MFL (manual fault localization) to augment the accuracy of fault localization. If fault localization applies high-level FL skills, such as those of expert developers, accuracy improves.

To study the various levels of MFL accuracy, we conducted a simulation case study. Specifically, we simulate the behavior of developers, inferring the suspiciousness of the fault for statements.

The MFL procedure we used in the fault localization phase in APR is as follows:

Step1: Developers select a statement from the faulty method as the cause of fault (*fault candidate statement*). The selected statement is the target for the APR to generate patches.
Step2: Developers apply a patch generation phase in an APR method to the fault candidate statement. The APR method then generates patches.
Step3: If the APR method does not generate any correct patches, developers select another statement.
Step4: Developers repeat Step2 and Step3 until a correct patch is generated, or all statements in the faulty method are selected.

We refer to one execution from Step1 to Step3 as an *attempt*.

It should be noted that, in our simulation, MFL has two assumptions: developers already know the faulty method and the cause of the fault originates from one statement. The first assumption is reasonable in practice because each method often has a corresponding test method to conduct a unit test. The second assumption is a limitation of our case study because if we considered multi-statement faults, the number of combinations of localized statements outstandingly increases making it difficult to execute our simulation.

In this paper, as described in Section I, we quantify the FL skill as the number of attempts developers execute before a correct patch is generated. The number of attempts corresponds to the number of statements selected as fault candidate statements. Hence, if MFL reduces the number of candidate patches generated by the APR until many attempts, even developers who do not accurately select the cause of the faults reduce the execution cost of APR.

Regardless of the fault localization results, some faults may not be fixed owing to the insufficient modification ability of the APR method. For such faults, it is difficult to compare MFL and automated fault localization. Hence, in this paper, we only used faults that can be fixed by the studied APR method if the faulty statement is localized.

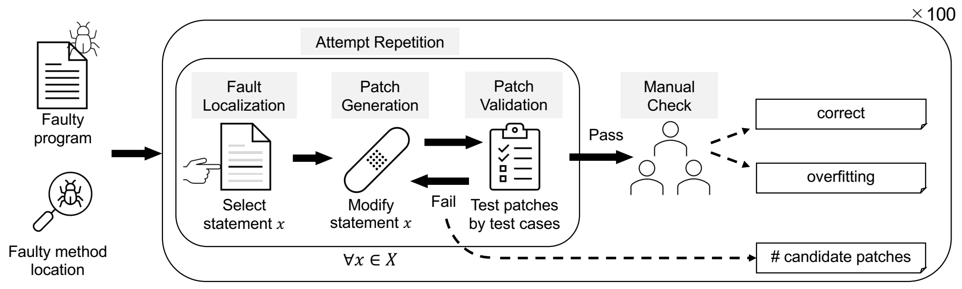


Fig. 1: Simulation overview of MFL

IV. Case Study Setup

A. Overview

The selected statement in MFL depends on the developers' skills. Hence, we simulate which statements the developers select and in what order. In our simulation, we compare MFL with a prior fault localization technique, SBFL. Figure 1 shows the simulation overview of MFL. Our simulation consists of two phases: *attempt repetition* and *manual check*. Also, the attempt repetition phase includes three processes: *fault localization*, *patch generation*, and *patch validation*.

As described in Section III, the number of attempts in MFL is the FL skill. Hence, we determine the number of attempts that is the simulated FL skill of developers before executing the simulation. The number of attempts ranges from one to the number of statements in the method. We study all the numbers within the range to simulate various levels of the FL skill. After determining the number of attempts, we execute the simulation.

B. Phases in the simulation

Attempt Repetition. In the simulation, one execution of this phase corresponds to an attempt of MFL. We repeat the execution of the attempt until the number of repetitions reaches the number of attempts that we determined before this simulation. Each execution of the attempt includes the following three processes.

1) *Fault Localization.* We select a statement in the method. The number of attempts is the simulated FL skill. Hence, in this process, we select the faulty statement at the final attempt while randomly selecting the other statements in the other attempts.
2) *Patch Generation.* We apply the studied APR method, TBar [4], to the selected statement to generate a set of patches. TBar is a recent template-based APR method. If developers can find the faulty statement perfectly (i.e., assuming perfect fault localization), TBar results in a record performance in the prior APR methods on Java [4]. Hence, TBar is suitable to evaluate MFL without the effect of repairing faults.
3) *Patch Validation.* We apply each of the generated patches to the method and execute all the test cases on each method a patch is applied. If the method fails to execute any test cases, apply another patch to the method and execute all the test cases. If there are no patches that pass the all test cases, we go back to the patch generation process. This repetition continues until the method passes all the test cases, the number of generated candidate patches reaches 10,000, or all the possible patches are generated.

Manual Check. APR can only check if the patch passes all test cases or not. Hence, we manually inspect all plausible patches by comparing the patch against the developer-provided patch that is available in the studied dataset. We classify the plausible patches into *correct patches* and *overfitting patches* by manual inspection. The correct patches are the patches that fix the corresponding faults semantically (i.e., the patches to make the same modifications as developer-provided). The overfitting patches do not fix the corresponding faults semantically, although they pass all the test cases. For example, if the generated patch is *a != b*, and the developer-provided patch is *!(a == b)*, then the patch *a != b* is classified as a correct patch because the two are semantically identical. The two authors decide whether they are overfitting or correct and classify the patches. The kappa coefficient between their evaluations is 0.89. Any patches of disagreement between them are discussed and decided with the other author.

As we randomly select statements in the fault localization phase except for the last attempt, we repeat the above two phases 100 times to improve the validity of the results. We evaluate MFL in terms of the number of candidate patches, the number of correct patches, the number of overfitting patches, and the number of attempts. We describe the details of the evaluation criteria in Section IV-D.

C. Details of the fault localization process

Because the fault localization process is the most important to simulate MFL, we describe the fault localization process more clearly. We define $S = \{s_1, s_2, ..., s_{bug}, ..., s_n\}$ as the set of statements in the faulty method. s_{bug} is the faulty statement of this method. The others are the statements that do not cause faults (*innocent statements*). We also define X as the subset of S that contains s_{bug} (i.e., $X \subseteq S$, $s_{bug} \in X$). We use this X to simulate the FL skill where developers successfully localize the faulty statement at the $|X|$th attempt. $|X|$ is the size of X (i.e., the number of statements contained in X). Hence, the meaning of X is the selected statements for $|X|$ attempts in MFL. For example, if $|X|$ is one (i.e., $X = \{s_{bug}\}$), we simulate the case where developers select the faulty statement at the first attempt; in other words, developers do not select any innocent statements. If $|X|$ is three (e.g., $X = \{s_2, s_5, s_{bug}\}$), we simulate the case where developers select the faulty statement at the third attempt after selecting innocent statements as the fault candidate statement two times.

It should be noted that we do not consider the order of selection except for the s_{bug}. This is because the number of candidate patches for each statement is constant, and the total number of candidate patches for an X remains the same even if the order of selection is changed. For example, the order of selection: $s_2 \rightarrow s_5 \rightarrow s_{bug}$ and $s_5 \rightarrow s_2 \rightarrow s_{bug}$ are indistinguishable in terms of the number of candidate patches. The lower $|X|$ value corresponds to the better fault localization performance, and developers need to localize the faulty statement accurately. We refer to $|X|$ as the *number of selected statements (NSS)*, a.k.a. the number of attempts. In this study, NSS simulates the FL skill.

For each NSS, the selection of innocent statements should affect the execution cost of APR. This is because the APR method generates a different number of candidate patches for each innocent statement. If developers select a statement in which the APR method generates many candidate patches, the generating and validating time would be longer. To simulate the impact of MFL on the execution cost of APR, it is necessary to simulate the selections of innocent statements for each NSS as well. Hence, as described in Section IV-B, we randomly select innocent statements 100 times for each NSS. For example, if NSS is ten, we select 100 different sets of nine innocent statements.

D. Evaluation criteria

Prior studies [3], [17] used the number of generated candidate patches until the valid patches (i.e., plausible patches) were generated to evaluate the execution cost of APR. This metric is independent of the execution environment and can be used to evaluate the execution cost of APR. Hence, we use the number of candidate patches (*NCP*) as our evaluation criterion. Particularly, *NCP* is the number of candidate patches for each NSS. As in the general case, the lower the *NCP*, the better the execution cost of APR.

We compare *NCP* between MFL and SBFL. To compare MFL with SBFL, we apply the APR method with SBFL to the faulty method and measure the *NCP*. Because SBFL results in the same *NCP* for all NSSs in a faulty method, we compare the *NCP* values of MFL for each NSS with the fixed *NCP* values of SBFL.

We also use the correctness of the plausible patch to evaluate MFL. We use the number of correct patches and overfitting patches classified by *manual check* phase to evaluate the correctness. Particularly, we compute the ratio of the correct and overfitting patches for each NSS for 100 repetitions.

E. Studied dataset

For the dataset, we select Defects4J [5], which is used to evaluate TBar and is the most popular dataset in the field of APR for Java programs. Defects4J includes faulty programs (faults) with their test cases and patches written by developers that repair the associated faults. Researchers have been using the Defects4J dataset to study APR to generate patches that are similar to those written by developers to repair the faults.

TABLE I: The number of studied faults retrieved from the Defects4J dataset

Project	Chart	Closure	Lang	Math	Mockito	Time	Total
faults	6/26	6/133	6/65	10/106	0/38	1/27	29/395

* Each cell in the second row shows x/y : x is the number of faults that are repaired correctly by changing one statement; y is the number of all faults in the Defects4J dataset for each project.

From the Defects4J dataset, we retrieve the studied faults that meet the following two conditions.

1) The faults that TBar repairs correctly.
2) The faults that are repaired by changing one statement.

We do not use the non-retrieved faults because MFL assumes that faults are only caused by one statement and we need to compare MFL and SBFL without the effect of the accuracy of TBar. Table I summarizes the number of retrieved faults.

V. Result

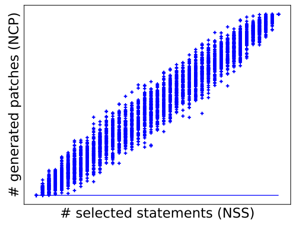
Based on whether MFL is better than SBFL, we classified the result into two patterns (A and B). Figure 2 shows examples of each pattern. Each figure corresponded to the result of a faulty method. The horizontal axis represented the number of selected statements (i.e., NSS) in MFL. The vertical axis is the number of candidate patches (i.e., *NCP*) for each NSS. Each dot represents the result of the execution of APR with MFL out of 100 executions for each NSS. The blue dots show the case where only a correct patch is generated. The red dots indicate both overfitting and correct patches are generated in the execution. Each NSS includes 100 dots. The horizontal line indicates the case in which a correct patch is generated by APR with SBFL. This result was a line instead of a dot because APR with SBFL resulted in the same *NCP* for all NSSs, as described in Section IV-D.

Pattern A. Pattern A (Figure 2a) indicates the case where SBFL is better or equal to MFL. The SBFL resulted in the lowest *NCP* compared to MFL, meaning SBFL easily detected the faulty statement for generating the correct patch.

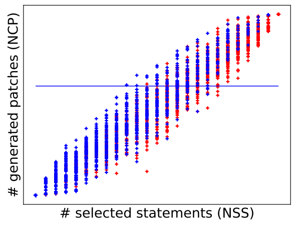
Pattern B. Pattern B (Figure 2b) implies that MFL is better than SBFL in a certain case. We observed some blue dots are lower than the horizontal line of SBFL. Hence, MFL resulted in a lower *NCP* than that of SBFL when NSS was lower than a certain value. The pattern also included the case where SBFL generated overfitting patches but none were correct. In both cases, MFL performed better than SBFL for any NSS.

We used the correctness of the plausible patches to decide the appropriate NSS where MFL is better than SBFL (i.e., the least FL skill to reduce the execution cost of APR). The appropriate NSS is the largest NSS and satisfy the following two conditions.

- The *NCP* value of SBFL was larger than the median *NCP* value of MFL. For example, in Figure 2b, we selected



(a) SBFL is better than MFL (A)



(b) MFL is better than SBFL (B)

Fig. 2: Example of simulation results: Pattern A (left) and Pattern B (right)

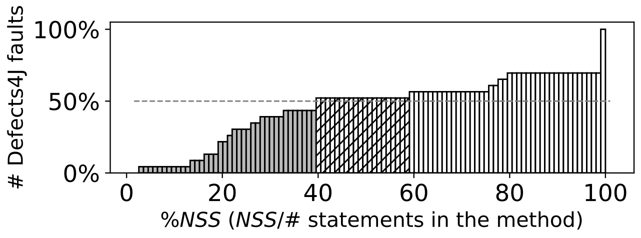


Fig. 3: The cumulative distribution of *NSS* for pattern B

all NSSs where more than half of the dots are under the horizontal line.

- There are more cases where only correct patches are generated than when overfitting patches are generated. For example, in Figure 2b, we selected all NSSs where more than half of the dots are blue.

For 75% of the studied faults, MFL may be superior to SBFL in APR. The number of faults classified as pattern A was seven, and the number classified as pattern B was 22. The faults in pattern A are almost evenly distributed for each project (i.e., one fault from the Chart project and two faults from the Closure, Lang, and Math projects). The percentage of faults classified as pattern A is 25%. Hence, there is 25% of faults in which SBFL is better than MFL. However, the percentage of faults classified as pattern B is 75%. For such faults, MFL was better than SBFL depending on NSS. Hence, MFL has the potential to reduce the execution cost of APR in the majority of faults.

The appropriate NSS of MFL is less than or equal to 40% of the number of statements in the method. Figure 3 shows the cumulative distribution for the number of faults in which MFL is better than SBFL for each *%NSS* calculated by dividing NSS by the number of statements in the method. We used %NSS instead of NSS because the number of statements for each method is different. Here all faults were classified as Pattern B. The histogram with diagonal lines represents the cumulative proportion that reached 50% (i.e., being in the majority). We also observe that MFL is better than SBLF in half of the faults in which %NSS is less than or equal to 40%. The result showed that developers need to specify the faulty statement before selecting 40% of the statements in the method in the MFL case. This value is acceptable for developers provided that they understand the target program. This is because it may not be difficult to localize faults before inspecting almost half of the statements.

TABLE II: The average number of patches per statement type

Statement Type	#Patch	Frequency
MethodCall	98.41	117
Binary	79.03	92
Name	73.25	16
Assign	62.78	46
VariableDeclaration	43.24	131
Return	32.88	78
Unary	23.40	25
Break	17.80	10
Throw	3.89	9
Continue	1.00	4

Finding

MFL has the potential to be superior to SBFL in 75% of the studied faults. To reduce the execution cost of APR with MFL, developers should select the faulty statement before selecting 40% of the statements in the method for half of the studied faults.

VI. Discussion

While conducting our experiment, we observed that different selected statement types led to distinctly different *NCP*s. To confirm that this observation was correct, we studied statement types, and their corresponding number of patches in the target programs. Table II shows the average number of patches per statement type. The #Patch column is the average number of patches generated from a single statement, classified as the type in the Statement Type column. The Frequency column is the total number of statements classified as the type in the Statement Type column. We used JavaParser[1] to classify the statement types. Note that a single statement can have more than one type. For example, a statement `a=method(b);` includes both the Assign type and the MethodCall type. Here, we used the type of child node of the Expression type from the statement parsed by JavaParser. If the statement does not have the Expression type, we used the name of the Statement class (e.g., Continue). The above example statement is classified as an Assign type. For more details of the statement types, please refer to the documentation [18].

The execution cost of MFL varies nearly 100 times depending on the selected statement. When using TBar, the number of patches for the "MethodCall" statement is nearly 100, but 1 for the "Continue" statement. That is, when using TBar-based MFL, the execution time may differ by a factor of close to 100, depending on the selected statement. This difference in the number of patches between statements is due

[1]https://javaparser.org

to APR, not MFL; and therefore, this difference is a side effect of using APR. SBFL does not remedy this side effect because it is a fully automated method. Fortunately, using MFL has the potential to remedy it because developers can avoid selecting the types of statements for which APR generates more patches.

From the result above, MFL may be particularly useful in supporting experienced developers. Experienced developers can find a faulty statement more accurately, and are less likely to select statements that do not contain fault but generate many patches. Hence, for experienced developers, fixing faults can be streamlined using MFL.

VII. Threats to Validity

Internal Validity. We assume that only one of the statements in the faulty method causes the fault. Hence, MFL can only be applied to such methods. Future studies are necessary to expand MFL to faulty methods that include multiple statements that cause faults.

Each developer has different knowledge, experiences, and skills. We can simulate MFL more rigorously by considering these differences. However, thid work aims to reveal a criterion that distinguishes whether using MFL is better than using automated fault localization techniques in terms of the execution cost. Hence, the distribution of FL skills is out-of-scope in this paper. Future studies are necessary to collaborate with developers and evaluate the performance of MFL by considering the differences.

To use MFL, developers should decide the fault candidate statement by manual effort. On the contrary, SBFL is fully automated. Future studies are necessary to evaluate the difference in effort between MFL and SBFL.

External Validity. We use TBar as our APR method in this work. Many APR methods have been proposed so far. Hence, different APR methods may show different results. Additionally, we use the Defects4J dataset only. Hence, future studies should evaluate the difference across different APR methods and datasets.

VIII. Conclusion and Future Work

In this paper, we investigated the number of candidate patches and set the following central question:

> *How many statements are developers allowed to inspect to reduce the execution cost of APR when using the FL skill?*

Through a simulation study, we found a criterion that established whether using the FL skill is better than using automated fault localization techniques. Developers reduced the execution cost of APR when they selected the faulty statement before selecting 40% of the statements in a Java method. We further found the number of candidate patches varied depending on the inspected statement. This is a side effect of APR, which varies the execution cost. SBFL does not remedy this side effect, whereas MFL may remedy it because MFL is not a fully automated method.

In the future, we would like to expand the investigated datasets and experiment using the skills of actual developers. In this paper, the developer's skill was quantified as the number of inspected statements before localizing the faulty statement from a method, but other skills may also be involved in the execution cost of APR with MFL. Hence, we would like to clarify the feasibility of MFL in an actual software development when considering the skills of actual developers. The replication package can be found here: https://www.dropbox.com/s/fotxbb7qh0fdxy2/compsac_2022_APR.zip?dl=0

Acknowledgments

This research was partially supported by JSPS KAKENHI Japan (Grant Numbers: JP18H04097, 21H04877, 21K17725) and JSPS International Joint Research Program with SNSF (Project "SENSOR": JPJSJRP20191502).

References

[1] J. S. Collofello and S. N. Woodfield, "Evaluating the effectiveness of reliability-assurance techniques," *J. of Syst. and Soft.*, vol. 9, no. 3, pp. 191–195, 1989.

[2] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *Trans. on Soft. Eng.*, vol. 45, no. 1, pp. 34–67, 2019.

[3] Y. Qi, X. Mao, Y. Lei, and C. Wang, "Using automated program repair for evaluating the effectiveness of fault localization techniques," in *Proc. of ISSTA*, 2013, pp. 191–201.

[4] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proc. of ISSTA*, 2019, p. 31–42.

[5] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proc. of ISSTA*, 2014, pp. 437–440.

[6] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," *J. of Syst. and Soft.*, vol. 82, no. 11, pp. 1780–1792, 2009.

[7] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *Trans. on Soft. Eng.*, vol. 42, no. 8, pp. 707–740, 2016.

[8] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proc. of ICSE*, 2012, pp. 3–13.

[9] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proc. of the POPL*, 2016, pp. 298–312.

[10] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandè, "Avatar: Fixing semantic bugs with fix patterns of static analysis violations," in *Proc. of SANER*, 2019, pp. 1–12.

[11] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proc. of ISSTA*, 2018, p. 298–309.

[12] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus, "Automatic repair of buggy if conditions and missing preconditions with smt," in *Proc. of CSTVA*, 2014, p. 30–39.

[13] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proc. of ESEC/FSE*, 2015, p. 532–543.

[14] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proc. of TAICPART*, 2007, pp. 89–98.

[15] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization of test information to assist fault localization," in *Proc. of ICSE*, 2002, pp. 467–477.

[16] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. L. Traon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," in *Proc. of ICST*, 2019, pp. 102–113.

[17] K. Liu, S. Wang, A. Koyuncu, K. Kim, P. Wu, J. Klein, X. Mao, Y. L. Traon, T. F. Bissyandé, and D. Kim, "On the efficiency of test suite based program repair : A systematic assessment of 16 automated repair systems for java programs," in *Proc. of ICSE*, 2020, pp. 615–627.

[18] JavaParser Contributors. (2021) javaparser-core 3.23.1 API. https://www.javadoc.io/doc/com.github.javaparser/javaparser-core/3.23.1/index.html.