Visualization of automated program repair focusing on suspiciousness values

Naoki Tane, Yusaku Ito, Hironori Washizaki, Yoshiaki Fukazawa
Department of Fundamental Science and Engineering
Waseda University
Tokyo, Japan
n.tane0228@fuji.waseda.jp

Abstract–Automated program repair (APR) can realize efficient debugging in software development. Automated program corrections using genetic algorithms (GA) can repair programs, including those with multiple bugs, but the repair process of GA-based APR is difficult to understand using logs because many modification program codes are generated. Consequently, Matsumoto et al. implemented a methodology for visualizing the process. Their proposed methodology provides an intuitive understanding of the conformance values (test case pass rates), generations, states, and operations performed to generate each variant; however, it lacks sufficient information to analyze whether defect localization is appropriate in APR. Herein we propose a new methodology to visualize the impact of fault localization on program evolution in GA-based APR and create a new tool. Additionally, a case study demonstrates the effectiveness of the proposed methodology and future works are considered.

Keywords–Visualization; Genetic Algorithm; Automated Program Repair; Fault Localization; Bug Localization

1 Introduction

Automated program repair (APR) is a technique that removes bugs without human intervention. APR outputs a bug-free program when given a buggy program and test suites.

kGenProg, which is a Java programmatic implementation of genProg, is a tool that uses genetic algorithms (GAs) for APR (GA-based APR)[1]. A key feature of kGenProg is its high portability. Users can easily change its parameters because kGenProg has an adaptable fault localization framework. However, it is difficult to analyze the repair process using the logs alone because kGenProg generates multiple programs during the modification process.

Macaw is an open-source software to visualize the evolutionary process of programs by kGenProg[2]. Macaw shows the code genealogy as a bird's eye view tree structure and detailed variant information, where a variant is a program generated in the evolutionary process. Its visualization provides an intuitive understanding of each variant. This information can be used to adjust the kGenProg parameters. One shortcoming is that Macaw does not provide enough information to analyze whether the fault localization is appropriate.

To address this issue, we propose a new methodology to visualize the impact of fault localization on the evolution of programs in kGenProg. Specifically, we create a new tool called Grackle based on this methodology and evaluate the visualization efficiency via a case study.

2 Background and problem

2.1 Fault Localization

Fault localization methods identify faulty program lines based on information obtained from the success or failure of test cases. A common category is spectral-based methods[3]. Spectral-based methods assign suspiciousness values according to a program statement's likelihood of a flaw. Tarantula[4], Ample[6], Jaccard[5], Ochiai[5], and Zoltar[7] represent Spectrum-Based Fault Localization(SBFL) to calculate the suspiciousness values (Table 1). The effectiveness depends on the given test case and program.

DOI reference number: 10.18293/SEKE2022-159

Table 1. Formulas to calculate the suspiciousness values for select SBFL methods

SBFL	formula for calculating the suspiciousness value
Ample	$Suspiciousness = \left\| \frac{a_{ef}}{a_{ef}+a_{nf}} - \frac{a_{ep}}{a_{ep}+a_{np}} \right\|$
Jaccard	$Suspiciousness = \frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$
Ochiai	$Suspiciousness = \frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf}) \times (a_{ef}+a_{ep})}}$
Tarantula	$Suspiciousness = \frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ep}}{a_{ep}+a_{np}} + \frac{a_{ef}}{a_{ef}+a_{nf}}}$
Zoltar	$Suspiciousness = \frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+10000 \times a_{nf} \times \frac{a_{ep}}{a_{ef}}}$

a_{ep}: Number of successful tests that executed the line
a_{ef}: Number of failed tests that executed the line
a_{np}: Number of successful tests that did not execute the line
a_{nf}: Number of failed tests that did not execute the line

2.2 APR

Many researchers have actively investigated technologies to increase the efficiency of debugging because debugging accounts for half of system implementation and testing costs[8]. These studies often focus on APR. APR removes bugs from buggy programs without human intervention[9]. It works by taking a buggy program and test cases as input and outputs a bug-free program.

2.3 kGenProg

GenProg is an APR methodology based on source code reuse[10]. kGenProg is an APR tool written in Java. It is a reimplementation of GenProg, which automatically repairs bugs using GAs. kGenProg works as follows. First, it infers the bug's line using a fault identification technique. Second, it generates multiple variants by modifying the lines that contain bugs. This step has two primary operations. One is mutation. Mutation means making minor changes to a selected variant to create a new variant such as insert, delete, or replace. The other is crossover. Crossover mixes two selected variants to generate a new variant. Third, unit tests are run on the generated variants. A repaired program is outputted if a variant passes all the tests. If not, kGenProg selects some of the generated variants and generates new variants based on them. Selection means that kGenProg takes some variants from the latest generation. kGenProg decides which variants to select by the pass rate of the unit test. Additionally, it selects some variants from the previous generation to account for the possibility of a poor pass rate for all generated variants.

2.4 Problem in use and existing visualization

Macaw visualizes the evolutionary process of programs with kGenProg. It was developed by Matsumoto et al.. Macaw's visualization provides an intuitive understanding of each variant's fitness value (test case pass rate), generation, state, and operation (insertion, deletion, replacement, crossover, or copy). Macaw helps adjust the parameters of the APR. However, it does not provide sufficient information to comprehend the impact of failure localization on the APR in the program's evolution. To address this shortcoming, we propose a new methodology to visualize the impact of fault localization on program evolution in kGenProg and evaluate the effectiveness of our methodology. This study aims to answer the following two research questions (RQs):

RQ1: Does the proposed methodology facilitate the understanding of fault localization?

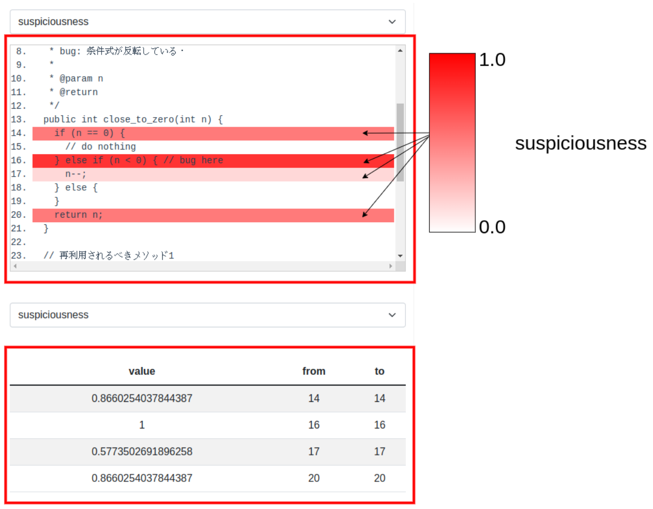
RQ2: Is the effect of fault localization in GA-based APR easily understood?

3 Visualization of code genealogy with fault-localization results

Similar to the Macaw project, our visualization displays the evolutionary process of kGenProg with the code genealogy shown as a bird's eye view tree structure and detailed variant information. These two factors can explain the representation of the alleged values. The color intensity represents suspicious values set for the row that operated to generate each variant. The color intensity also represents each row that is subject to the suspicious values of each variant. Figure 1 depicts the visualization of the suspiciousness values in the detailed information of the variant. Our visualization method provides an intuitive recognition of the suspiciousness value for the lines in the bug framework. The source code of each variant is displayed at the top of the detailed information, and the suspiciousness is indicated in red. The darker the red color, the higher the suspiciousness value in the line.

Figure 2, which shows the flow to generate child variants, overviews the proposed visualization of the suspiciousness values for the code genealogy. First, fault localization calculates the suspiciousness values of each line of the parent variant. This is similar to how kGenProg generates variants. Second, the suspiciousness value set at the line where kGenProg is performed is identified by the color intensity of the edge of the tree structure in the bird's eye view.

Figure 3 shows the visualization of the suspiciousness value for the code genealogy. Similar to the visualization proposed by Macaw, each node represents a single variant,



```
8.     * bug: 条件式が反転している・
9.     *
10.    * @param n
11.    * @return
12.    */
13.   public int close_to_zero(int n) {
14.     if (n == 0) {
15.       // do nothing
16.     } else if (n < 0) { // bug here
17.       n--;
18.     } else {
19.     }
20.     return n;
21.   }
22.
23.   // 再利用されるべきメソッド1
```

value	from	to
0.8660254037844387	14	14
1	16	16
0.5773502691896258	17	17
0.8660254037844387	20	20

Figure 1. Proposed method to visualize the suspiciousness values in the detailed information of the variant

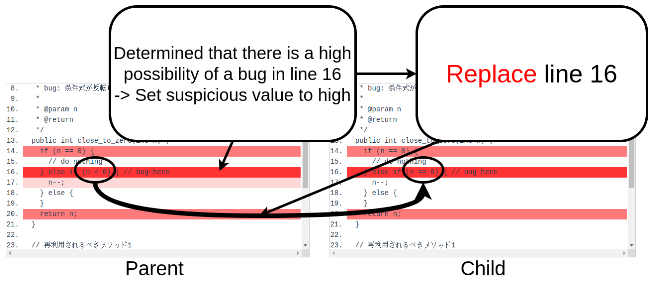

Determined that there is a high possibility of a bug in line 16 -> Set suspicious value to high
Replace line 16
Parent
Child

Figure 2. Flow of the generation of child variants

and nodes on the same y-axis mean that they are from the same generation of kGenProg. Circle nodes represent newly generated variants, and small circle nodes denote variants copied from all ages. Crosses indicate variants that failed to compile or invalid variants. The proposed visualization describes the number of nodes in each generation below the cross. The adaptive value is an indicator of the pass rate. In our method, the darker the green color, the higher the adaptive value.

Unlike Macaw, the red intensity of the edge of the tree structure in the bird's eye view of our method represents the suspiciousness value set at the point where kGenProg operated (insertion, deletion, or replacement). The darker the color, the higher the suspiciousness value of the changed part. It should be noted that edges representing crossover and copy operations are not colored because the crossover operation combines codes of two variants and the copy operation in kGenProg does not make any changes. Hence, areas with high suspiciousness values are easily visualized and the influence of fault localization on APR can be understood intuitively. Similar to Macaw, clicking on a node shows details of the generated variants.

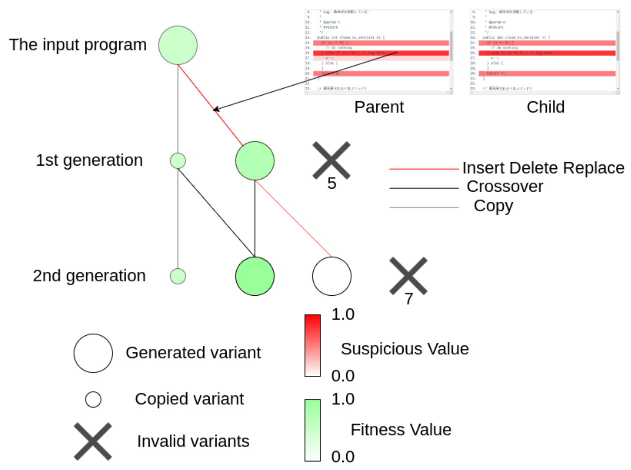


Figure 3. Visualization of the suspiciousness value for the code genealogy

4 Tool implementation

To implement our methodology, kGenProg must be modified. By default, kGenProg has an option to output JSON files. In addition to JSON files, the source code and suspiciousness values must also be outputted in JSON to implement our method. Therefore, we created a modified kGenProg to output the source code and suspiciousness values as JSON files. We also created a tool to implement our methodology. We call this tool Grackle. Grackle can visualize the suspiciousness values by reading the JSON file output from the modified kGenProg.

Figure 4 shows the flow using Grackle. First, our methodology builds a modified kGenProg, which outputs source code and suspiciousness values as JSON files and creates an executable file (JAR file). Second, our methodology runs the auto-fix in the executable file. Finally, Grackle is used to visualize the flow of the automated modification by inputting the JSON file output from the involuntary conversion.

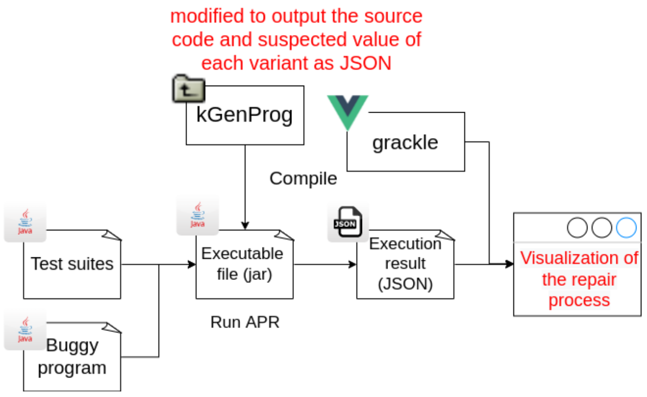
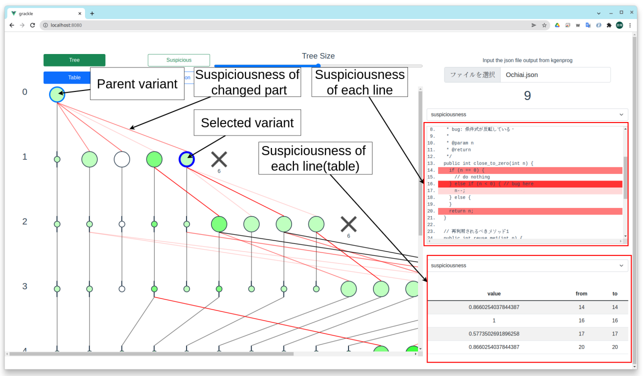

modified to output the source code and suspected value of each variant as JSON
kGenProg
grackle
Compile
Test suites
Executable file (jar)
Execution result (JSON)
Visualization of the repair process
Run APR
Buggy program

Figure 4. Flow of Grackle

Figure 5 shows a screenshot of the code genealogy visualized by Grackle. The left side shows the code genealogy, while the right displays detailed variant information. Grackle highlights the selected variant with a blue border and the parent variant with a light blue frame. Hence, users can visualize the suspiciousness value of each line of code in the variant. Additionally, the red intensity of the edge of the tree structure in the bird's eye view represents the suspiciousness value set at the changed line in the code system.


Tree
Table
Suspicious
Tree Size
Parent variant
Suspiciousness of changed part
Suspiciousness of each line
Selected variant
Suspiciousness of each line(table)
Input the json file output from kgenprog
ファイルを選択
Ochiai.json
9
suspiciousness

```
 * bug: 条件式が反転している・
 *
 * @param n
 * @return
 */
public int close_to_zero(int n) {
  if (n == 0) {
    // do nothing
  } else if (n < 0) { // bug here
    n--;
  } else {
  }
  return n;
}

// 再利用されるべきメソッド1
public int reuse_me1(int n) {
```

value	from	to
0.8660254037844387	14	14
1	16	16
0.5773502691896258	17	17
0.8660254037844387	20	20

Figure 5. Screenshot of the code genealogy visualized by Grackle

The source code differences and test results in Variant Details can be viewed by changing the mode. Hence, the edge types in Code Genealogy and Macaw can express various operations. Figure 6 shows an example of the visualization when changing the mode in Grackle. Figure 7 shows a screenshot.

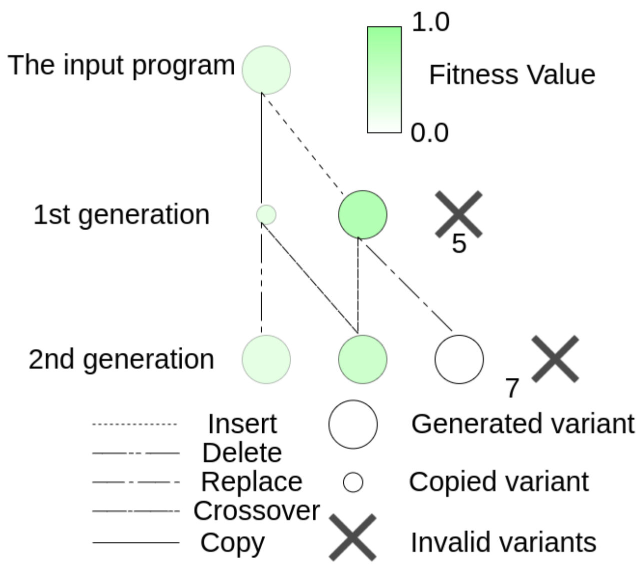
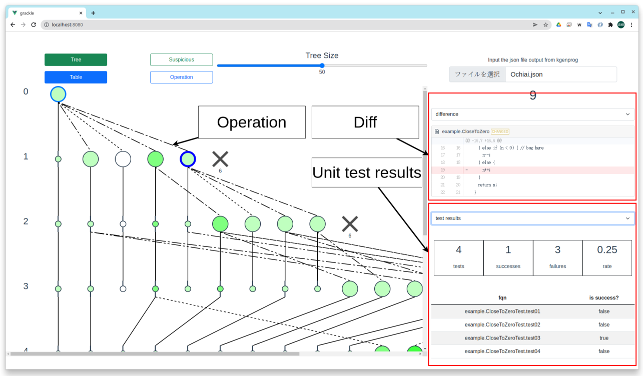


Figure 6. Example of the visualization when changing the mode in Grackle



```
@@ -16,7 +16,6 @@
16  16        } else if (n < 0) { // bug here
17  17          n--;
18  18        } else {
19      -       n++;
20  19        }
21  20        return n;
22  21      }
```

4	1	3	0.25
tests	successes	failures	rate

fqn	is success?
example.CloseToZeroTest.test01	false
example.CloseToZeroTest.test02	false
example.CloseToZeroTest.test03	true
example.CloseToZeroTest.test04	false

Figure 7. Screenshot of the code genealogy visualized by Grackle

We implemented Grackle using the Javascript framework Vue.js. Grackle can also run in modern web browsers such as Google Chrome, Firefox, Safari, and Microsoft Edge. This example handles a small log. A normal behavior is observed even for a relatively large record with 1970 variants and a generation number of 100. However, it does not support visualizations of the suspiciousness values for multiple java programs. This support should be addressed in the future. Clicking on a node in the bird's-eye view on the left shows detailed information about the variant on the right. Clicking on the button in the upper left corner changes the edges of the bird's eye view. The correct upper dialog allows users to select a JSON file representing the APR process output by kGenProg. Then users can select the code and the Diff representing the alleged value using the select box above the detailed information. They can also select a table of suspiciousness values and unit test results using the selection box below the detailed information.

5 Case study

5.1 Conditions

We performed an APR with kGenProg under the following conditions to verify whether our methodology is effectively implemented:

Program: kGenProg's QuickSort test program (QuickSort.java)
Fault Localization: Ochiai

We also examined the RQs by comparing Macaw and Grackle visualizations for APR under the above conditions. Macaw is a project to visualize the evolution process of programs using kGenProg, while Grackle is the tool devised in this study.

5.2 Results

Figure 8 shows a screenshot of Macaw. Figure 9 shows a screenshot of Grackle. Both Macaw and Grackle provide code genealogy to intuitively understand the test case pass rate, generation, status, and operation performed by kGenProg to generate each variant. Both also generate detailed information about the variants, allowing users to understand the differences between the before and after operations and the unit test results. In addition, Grackle provides the alleged value of the changes made. Grackle also shows the suspiciousness value calculated for each variant line by fault localization in the variant details.

5.3 RQ1: Does the proposed methodology facilitate the understanding of fault localization?

Macaw does not give information about the suspiciousness with bugs calculated by fault localization in each variant. On the other hand, Grackle provides an intuitive understanding of which line of code in each variant is responsible for the calculated suspiciousness value. Thus, the proposed method facilitates the understanding of fault localization.

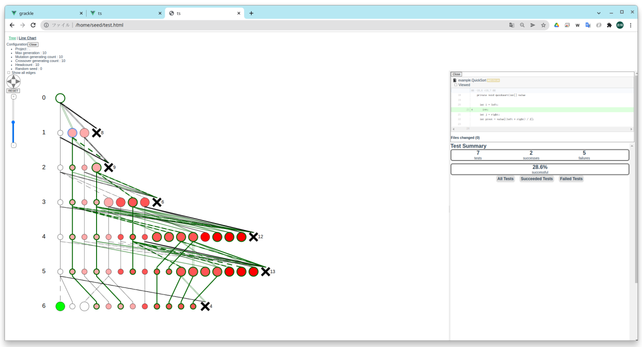


Figure 8. Screenshot of Macaw

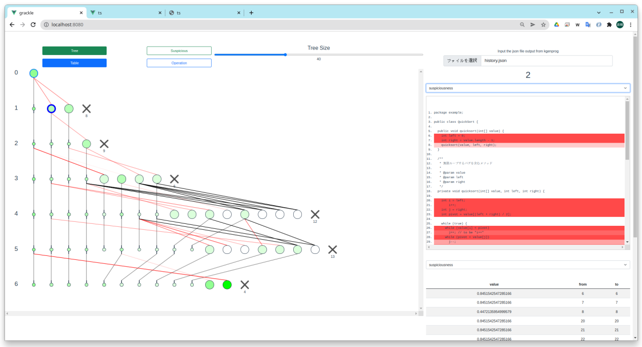


Figure 9. Screenshot of Grackle

5.4 RQ2: Is the effect of fault localization in GA-based APR easily understood?

Macaw's code genealogy does not provide information about the alleged value of the point of change in each operation. On the other hand, Grackle's code genealogy generates information about the alleged value of the point to be changed in each mutation operation. This information helps realize an intuitive understanding of APR's fault localization behavior using GAs. Thus, the proposed method clarifies the effect of fault localization.

6 Conclusion and future work

Herein we propose a methodology and tool to visualize the impact of fault localization on the evolution of programs in kGenProg. The effectiveness was evaluated via a case study. The case study qualitatively demonstrates the usefulness of our methodology and tool.

We visualize three prominent use cases: to compare different fault localization strategies, to select an appropriate fault localization strategy, and to realize an intuitive understanding of changes made by KGenProg. kGenProg can easily change fault localization strategies. The proposed methodology supports an intuitive understanding of the changed parts of the code using the code genealogy and the suspiciousness values. For example, using a test suite and a buggy program as inputs, different fault localization strategies (Ample, Jaccard, Ochiai, Tarantula, or Zoltar) can be applied and the APR run. The proposed methodology can compare fault localization strategies directly and elucidate the impact of automated corrections by strategy.

The proposed methodology can provide an understanding for setting suspiciousness values for fault localization. It can be employed to check the code genealogy and the details of the variant to verify that the suspiciousness value is set appropriately. For example, a fault localization strategy may negatively impact the APR if the suspiciousness value is set low for a buggy part or high for a non-buggy part. In this way, fault localization can be evaluated, allowing users to select an appropriate bug identification tool. This should improve the efficiency of APR.

Use cases assume that the fault localization strategy is working properly. Our methodology highlights significant changes made by kGenProg. The visualization of suspiciousness values using code genealogy edges shows changes with high suspiciousness values (i.e., where bugs are likely to be present). In this case, a bug is likely to be fixed accurately.

In the future, we plan to evaluate our methodology and tool quantitatively. We also plan to devise use cases for the proposed methodology and verify the practicality of our methodology and tool.

References

[1] Y. Higo, S. Matsumoto, R. Arima, A. Tanikado, K. Naitou, J. Matsumoto, Y. Tomida, and S. Kusumoto, "kGenProg: A High-Performance, High-Extensibility and High-Portability APR System," 2018 25th Asia-Pacific Software Engineering Conference (APSEC), Nara, Japan, 2018, pp. 697-698.

[2] Y. Tomida, Y. Higo, S. Matsumoto and S. Kusumoto, "Visualizing Code Genealogy: How Code is Evolutionarily Fixed in Program Repair?," 2019 Working Conference on Software Visualization (VISSOFT), 2019, pp. 23-27, doi: 10.1109/VISSOFT.2019.00011.

[3] J. Xuan and M. Monperrus "Learning to Combine Multiple Ranking Metrics for Fault Localizetion" 2014 IEEE International Conference on Software Maintenance and Evolution pp. 191-200 2014.

[4] J. A. Jones, M. J. Harrold and J. Stasko, "Visualization of test information to assist fault localization", Proceedings of the 24th international conference on Software engineering, pp. 467-477, 2002.

[5] R. Abreu, P. Zoeteweij and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization", Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION 2007. TAICPART-MUTATION 2007, pp. 89-98, 2007.

[6] V. Dallmeier, C. Lindig and A. Zeller, "Lightweight bug localization with ample", Proceedings of the sixth international symposium on Automated analysis-driven debugging, pp. 99-104, 2005.

[7] T. Janssen R. Abreu and A. J. C. van Gemund "ZOLTAR: A toolset for automatic fault localization" Proceedings of the International Conference on Automated Software Engineering (ASE'09) - Tool Demonstrations.

[8] Britton, T., Jeng, L., Carver, G., Cheak, P. and Katzenellenbogen, T. (2013). Reversible Debugging Software: Quantify the time and cost saved using reversible debuggers.

[9] YASUDA, Kazuya, ITOH, Shinji, NAKAMURA, Tomonori, HARADA, Masao, HIGO, Yoshiki. Automated Program Repair Using Donor Code Generation Based on Features of Targeted Systems. Computer Software. 2021, vol. 38, no. 4, p. 4_23-4_32.

[10] 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", ICSE'12, pp. 3-13.