Visualizing Code Genealogy
— How Code is Evolutionarily Fixed in Program Repair? —

Yuya Tomida, Yoshiki Higo, Shinsuke Matsumoto and Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University, Japan
{y-tomida, higo, shinsuke, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—Automated program repair (in short, APR) techniques that utilize genetic algorithm (in short, GA) have a capability of repairing programs even if the programs require multiple code fragments to be changed. Those techniques repeat program generation, program evaluation, and program selection until a generated program passes all given test cases. Those techniques occasionally generate a large number of programs before a repaired program is generated. Thus, it is difficult to understand how an input program is evolved in the loop processing of genetic algorithm. In this paper, we are inspired by genealogy and propose a new technique to visualize program evolution in the process of automated program repair. We have implemented the proposed technique as a software tool for kGenProg, which is one of GA-based APR tools. We evaluated the proposed technique with the developers of kGenProg. In the evaluation, the developers found latent issues in kGenProg's processing and came up with new ideas to improve program generation. From those results, we conclude that our visualization is useful to understand program evolution in the APR process.**

***Index Terms*—Automated Program Repair; Code Genealogy; Evolution History; Visualization; Genetic Algorithm**

I. Introduction

Debugging takes considerable time and effort in software development. There is a research report that debugging occupies even over 50% of total development cost [1]. Thus, debugging support is a promising way to reduce development cost and achieve high reliability. A variety of debugging support techniques have been proposed, such as program slicing [2], automated test generation [3], and fault localization [4].

Automated program repair (in short, APR) is one of the most emerging and promising technique to facilitate debugging. The input of APR technique is a buggy program and a set of test cases. APR generates a repaired program without any human intervention. GenProg, which is an APR with genetic algorithm [5], is the most popular technique [6]. In the repairing process, GenProg generates multiple programs that are slightly changed from the input one and runs all test cases for each generated program. If none of the generated programs passes all the test cases, GenProg selected some of them and generated programs based on them again. GenProg repeats this process until a generated program passes all the test cases.

If GenProg cannot repair a given buggy program, researchers and practitioners want to know why the input program was not repaired and what they can do to repair it. By analyzing how the genetic algorithm proceeds for the given program, they might be able to find more appropriate parameters for GenProg or come up with some ideas to improve the APR technique. However, GenProg generates a large number of programs in the repairing process; just logging the process is not enough to support developers/practitioners to analyze the APR process.

In this paper, we propose a new methodology to visualize how the input program is evolved with genetic algorithm. Our methodology visualizes program evolution as genealogy. We have implemented a tool based on the methodology and applied it to repair real bugs. We interviewed developers of an APR tool with genetic algorithm to evaluate our methodology qualitatively.

II. Preliminaries

A. Automated Program Repair

APR is a technique that takes a buggy program and a set of test cases as its input and outputs a repaired program. Herein, a buggy program means a program that fails at least a test case and a repaired program means a program that passes all the test cases.

B. GenProg

GenProg is an APR technique with genetic algorithm. Although the primary objective of GenProg is to fix existing bugs of program, GenProg has a significant potential to generate a program from scratch [7]. In GenProg, each generated program is regarded as a variant. Each variant is represented as a list of genetic operators to generate it. Thus, an input program (an initial variant) includes an empty list. GenProg improves an input program by applying genetic operators repeatedly. Hereafter, we call a list of genetic operators to generate a variant base.

Figure 1 shows an overview of GenProg. Firstly, GenProg infers lines including the bug by utilizing a fault localization technique. Secondly, GenProg generates multiple variants by changing the inferred lines. Then, GenProg runs all test cases for each generated variant. If there is a variant that passes all the test cases, GenProg outputs it as a repaired program. If not, GenProg selects some of the generated variants and generates new variants based on them. GenProg repeats this process until a generated variant passes all the test cases. In GenProg, selection, mutation, and crossover are defined as follows:

- *Selection*: GenProg picks up some variants from all the variants in the latest generation. Picked-up variants are used for generating next-generation variants. GenProg has a fitness function for the selection. In the fitness function,

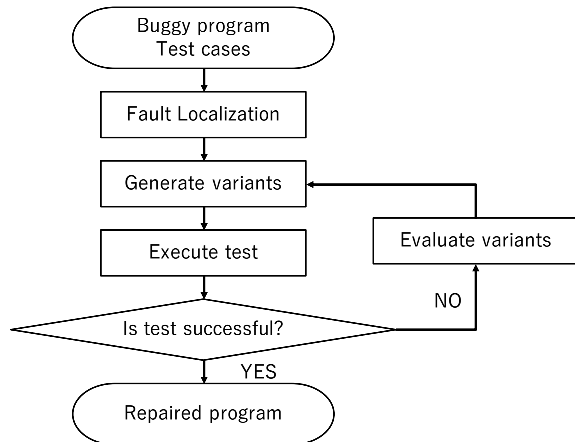


Fig. 1: Overview of GenProg

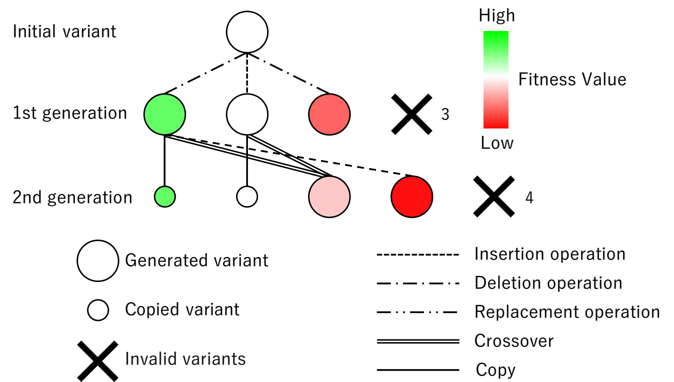


Fig. 2: Visualization example

variants that pass more test cases are regarded as better. GenProg remains some variants that were generated during previous generations because all variants in the latest generation might be worse than the variants in previous generations. We call remaining variant *copy* operation.

- *Mutation*: mutation means generating a new variant by adding a small change to a selected variant. A mutation operation is either of insertion, deletion, or replacement. In the case of insertion and replacement, GenProg utilizes an existing program statement in the initial variant.
- *Crossover*: crossover means generating a new variant by mixing bases of two selected variants. There are three types of crossover: single-point crossover, uniform crossover, and random crossover.

III. Proposed Visualization Method

A. Overview

In order to support to understand how GA-based APR works, we propose a visualization system of program evolution. Usually, GA generates a massive number of variants in an evolution series. Although evolution history can be represented as simple topology (i.e., tree structure), the history becomes complex and vast to understand. There are two key factors of the visualization. The first factor is to provide an overview of the evolution, namely, *code genealogy*. The code genealogy helps to answer the following questions. *How was the buggy program evolved? Which type of code generation was effective? How many generations were necessary to repair the program?* The second key factor is to help dig into details of each generated variant such as test results and what kind of operations were performed to generate the variant.

B. Visualization Strategy

The proposed visualization strategy is to represent a code genealogy as a tree structure for a bird's eye view. Furthermore, the detailed variant information is also shown according to user interaction. Figure 2 shows a concrete example of the proposed visualization. By scrolling down this view, users can grasp transition of code evolution.

Each node represents a single variant. Nodes located in the same Y-axis mean that they were generated in the same generation. A circle node means that a variant newly generated in the generation. A tiny circle shows that the node was copied from the previous generation. In other words, the shape means that the variant has already been generated before this generation. A cross represents either of two cases: a variant failed to be compiled or a variant coincidentally has the same source code as any of variants that have been generated in the past genealogy. In this paper, we call them *invalid variants* because of their compilation error and redundancy. The number of invalid variants is also shown in the side of the cross mark. This information helps to understand *which operation is likely to generate these invalid variants*. Color of circle nodes represents an increase or decrease in fitness value. White nodes mean that their variants have the same fitness value as the initial variant. Green means better and red means worse. This coloring helps APR users to grasp *does APR work well or not, intuitively*. The type of the ingoing edge of a node expresses what kind of operations was performed to generate the variant.

Details of generated variant can be confirmed by clicking each node. The detail includes diffs of source code and its test results.

C. Use Case

1) APR tool improvement: We introduce a use case scenario where an APR developer tries to develop a smarter generation operation. He/she might understand *does the new operation generate better variants* by checking overall generation history using our visualization. If edges of the operation are more likely to connect with green circles, it means the operation works well. For more detailed analysis, he/she can confirm *does the new operation work as he/she had expected* by checking diff of generated variants.

2) APR parameter adjustment: Here is another scenario where an APR user tries to repair a bug in his/her program. Parameter adjustment is a significant activity in such a situation. One important option for the adjustment is depth-first or width-first. Depth-first means that the number of max generations

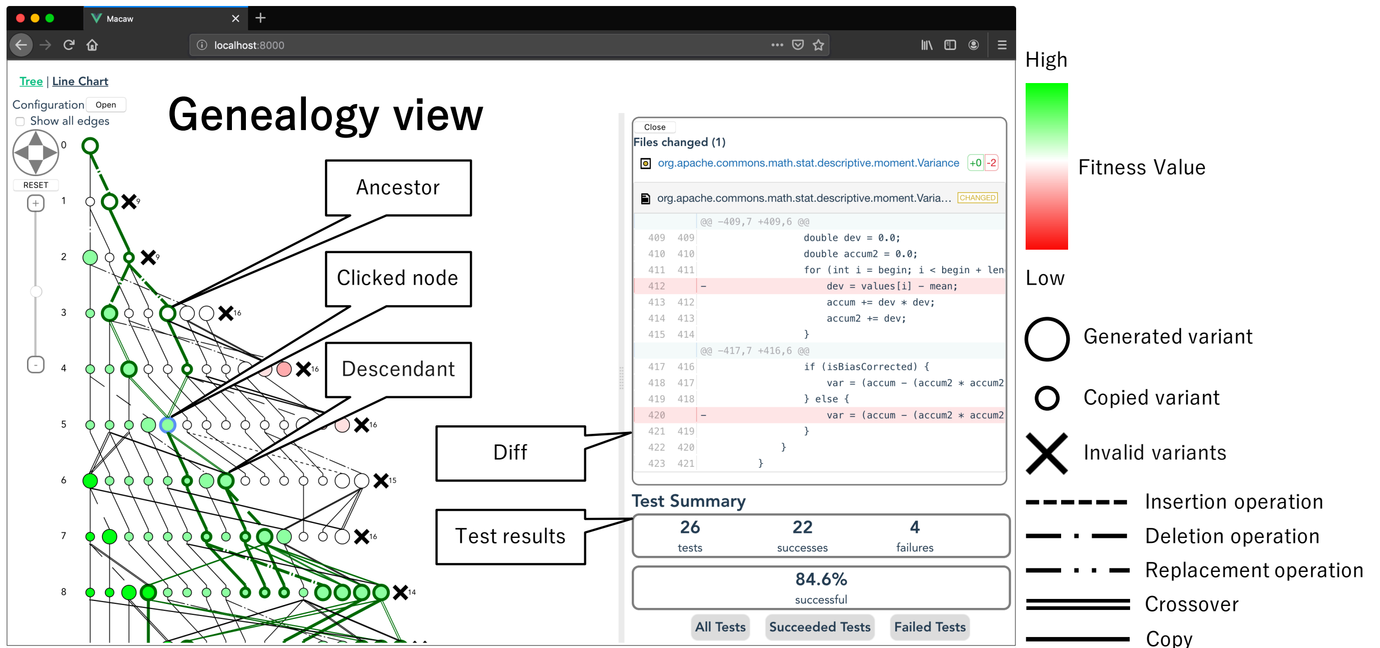


Fig. 3: Screenshot of code genealogy visualized by Macaw

(i.e., depth) to be large and the number of generated variants in a single variant (i.e., width) to be small. Depth-first would be an adequate choice when a repaired program fails many tests. In such a case, the repaired program is necessary to evolve with many generations. In contrast to that, when a program fails a single test, width-first might be a better choice. Our visualization helps APR users to choose these parameters to adjust. For example, let us consider a case where fitness value of generated variants seems unchanged during code evolution. This situation can be regarded as the bug may difficult to fix by applying some mutation operations. In other words, more mutation operations may be necessary to fix the bug. So, APR user should adjust parameters to be depth-first.

D. Implementation

We implemented our methodology and named the tool as Macaw[1]. Macaw is written by Javascript framework Vue.js[2] and runs on modern Web browsers such as Google Chrome, Firefox, Safari, and Microsoft Edge. SVG[3] is used to render nodes and edges. Macaw is open source software[4]. Figure 3 shows a snapshot of Macaw. The left pane shows code genealogy and the right pane shows the detailed variant information. The highlighted path shows ancestors and descendants of a clicked variant. It can be seen that the clicked variant was generated by applying crossover. Also, the variant had many children. The concrete diff information is also shown in the right pane. The variant is generated by just removing two statements. The bottom test summary pane gives further helpful information that 22 tests were passed but four tests were still failed. Which means, the variant is necessary to continue to evolve more.

[1]Macaw is an ARP Comprehension Assist tool With visualization
[2]https://vuejs.org
[3]Scalable Vector Graphics
[4]https://github.com/ty-v1/Macaw

IV. Case Study

To evaluate the usefulness of our visualization, we conducted a case study. In the case study, we visualized evolution of variants generated by kGenProg [8]. Our targeted bugs are ones of Apache Commons Math. Those bugs were collected from Defect4J [9]. The target bugs are shown in Table I. Generated code genealogy strongly depends on parameters such as the maximum number of generations, and the number of variants generated in each generation. In order to conduct a case study using various code genealogies, we executed kGenProg with various parameters. For example, Math95, which contains a single failed test, was executed under two situations: width-first (i.e., max generation = 300, number of variants = 10) and depth-first (i.e., max generation = 40, number of variants = 75). The total number of variants in both situations are the same. The participants include two professors, four graduate students, and an undergraduate student. All participants are developers of kGenProg.

A. Procedure

The case study was conducted with the following steps. Firstly, we explained to the participants how to use Macaw. Then, the participants used Macaw for a practice of using our visualization. The subject of this practice was not an actual but a tiny buggy example. This example can be fixed within three generations with less than 100 variants. These preparation steps took about 20–30 minutes. Finally, the participants discussed the target bugs. A time limitation was not set for the discussion. Each participant used Macaw freely.

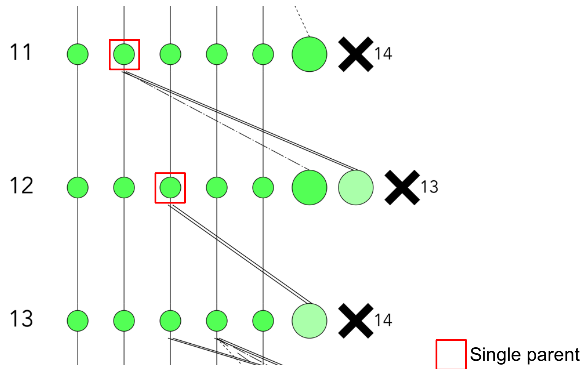


Fig. 4: Crossover bug found by Macaw

values of multiple variants were the same. The second bug was in the crossover. The number of variants used for the crossover must be two, but the number of some variants generated by the crossover was one. Nodes surrounded by red boxes in Figure 4 show the bug.

3) Suggestions for Visualization: Some participants gave positive suggestions for improvement of Macaw's visualization. Figure 5 shows two suggestions.

Figure 5a is to show a percentage of applied operations as a pie chart. Although Diff information shows the finest code differences, the percentage of applied operations is difficult to grasp. In the case of Figure 5a, the focused variant is generated by many deletion operations. This bug might be fixed by removing some statements.

Another idea is to visualize further useful metrics in each variant. The idea is shown in Figure 5b. In this case, readability and performance for each variant are visualized. Readability can be roughly measured by traditional complexity metric and code size. Also, performance can be measured by executing test cases. We can see that significantly low performance variants were generated in the second generation. It means that the overall APR performance will decrease by such slow variants. By visualizing this information, APR users can try to adjust parameters and to make better fitness functions.

B. Results

1) Suggestions for Improvement of kGenProg: Four suggestions were given by the participants. The first suggestion was how to select variants for crossover. In Math43, a solution was generated by crossover and test results between its parents greatly differed. Therefore, there was a suggestion that kGenProg would search the solution more efficiently if kGenProg selects variants whose test results are different from each other for crossover.

The second suggestion was that kGenProg did not utilize the feature of GA because kGenProg did not define the fitness value well. kGenProg defines the fitness value as the rate of test success, but most bugs in Defect4J fail only one test case.

Thirdly, there was a suggestion that the insertion operation of kGenProg would be improved if kGenProg insert multiple program statements at once. For example, inserting a statement including a variable reference and the statement including the definition of the variable is a typical case.

Finally, a suggestion for the insertion operation was obtained. The suggestion is that if many invalid variants were generated from a certain variant, the certain variant should be excluded from the target of the selection. The participants found that variants that have not been found yet were not generated from those variants in many cases. Related to that, they found that more invalid variants were generated and a variety of variants got poor as the generation proceeded. To solve this problem, the participants proposed a new fitness function that regards newer variants as better.

2) Bugs in kGenProg: Two bugs of kGenProg were found in the case study. The first bug was that older variants were unintentionally selected with higher priority when fitness

C. Discussion

In this section, we discuss why those suggestions were obtained.

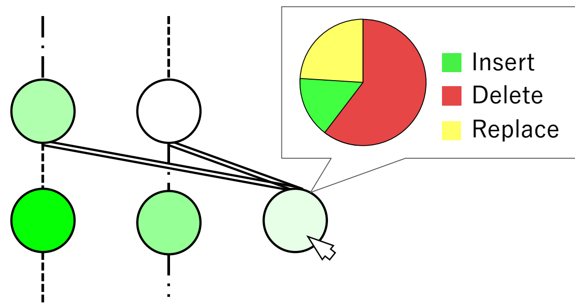
1) Suggestions for Improvement of kGenProg: A suggestion that kGenProg searches solutions more efficiently by using variants whose test results are largely different from each other were obtained because the participants can understand what kinds of operations were performed to variants by the operation visualization.

The participants found that the fitness value did not change when kGenProg generated the solution by the fitness value visualization. This means that GA did not work well. Therefore, they suggested that kGenProg did not define the fitness value well.

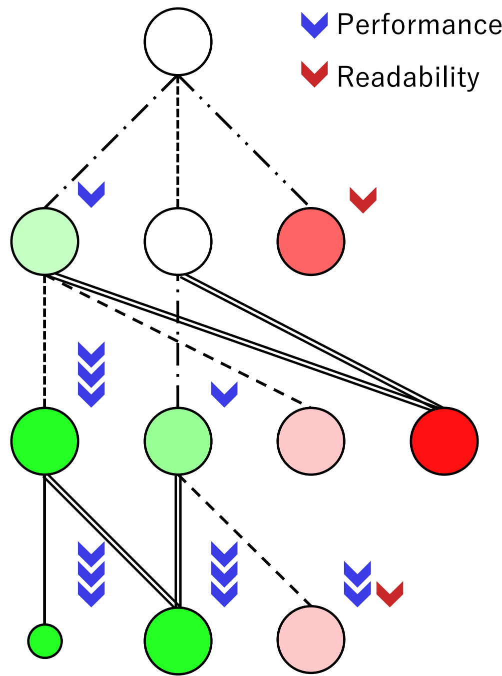
The participants easily found statements inserted by the insertion operation by the source code differences and the operation visualization. They found that most statements inserted by the insertion operation included references to undefined variables. This means that generated variants including such statements were invalid ones. Accordingly, they suggested that

TABLE I: Visualized code genealogies used in the case study

Bug ID	# failed tests	Max generation	# variants generated by mutation	# variants generated by crossover	# variants selected in every generation	Crossover type
Math95	1	300	10	0	5	No crossover
		40	75	0	5	No crossover
Math43	6	200	10	5	5	Random crossover
		40	55	5	5	Random crossover
Math2	1	400	2	40	20	Random crossover
		400	2	40	20	Uniform crossover
		400	2	40	20	Single-point crossover



(a) Visualizing a percentage of applied operations



(b) Visualizing useful metrics

Fig. 5: Suggestions of visualization improvement

the insertion operation of kGenProg would be improved if kGenProg inserts multiple program statements at once.

There were many copied variants in the visualized evolution. The participants found that more invalid variants were generated and a variety of variants got poor as the generation proceeded. Therefore, they suggested that if many invalid variants were generated from a certain variant, the certain variant should be excluded from the target of the selection.

2) Bugs in kGenProg: There were many copied variants and the parent-child relationship was represented by edges so that the participants found a bug that older variants were unintentionally selected with higher priority when fitness values of multiple variants were the same.

Related to the above, some variants had only a single crossover incoming edge. Therefore, the participants found a bug that only a single variant was used for crossover.

3) Summary: We conclude that our visualization is useful for developers of GA-based APR tools. Especially, kGenProg can be improved based on the suggestions that were obtained based on our visualization.

V. Conclusion

In this paper, we proposed a new visualization of how an input program is evolved in GA-based APR tools. We also conducted a case study with developers of a GA-based APR tool to evaluate our visualization. We obtained many opinions through the developers. The opinions include suggestions of the insertion operation of the tool and even latent bugs in the tool.

There are still many improvements for Macaw. For example, if Macaw visualizes how many times each operation such insert or crossover is applied to and how many invalid variants are generated by each operation, users can easily understand which operation is useful to generate a repaired program.

Acknowledgments

This work was supported by MEXT/JSPS KAKENHI 17H01725.

References

[1] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, "Reversible Debugging Software: Quantify the time and cost saved using reversible debuggers," 2013.
[2] M. Weiser, "Program Slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE '81. IEEE Press, 1981, pp. 439–449.
[3] L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 3, pp. 215–222, Sep. 1976.
[4] J. A. Jones and M. J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-localization Technique," in *Proceedings of the 20th ACM/IEEE International Conference on Automated Software Engineering*, 2005, pp. 273–282.
[5] 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 *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 3–13.
[6] K. F. Man, K. S. Tang, and S. Kwong, "Genetic algorithms: concepts and applications [in engineering design]," *IEEE Transactions on Industrial Electronics*, vol. 43, no. 5, pp. 519–534, Oct. 1996.
[7] K. Shimonaka, Y. Higo, J. Matsumoto, K. Naito, and S. Kusumoto, "Towards Automated Generation of Java Methods: A Way of Automated Reuse-Based Programming," in *Proc. of the 12th IEEE International Workshop on Software Clones*, Mar. 2018, pp. 30–36.
[8] 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," in *the 25th Asia-Pacific Software Engineering Conference*, Dec. 2018, pp. 697–698.
[9] 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.