Applying Multi-Objective Genetic Algorithm for Efficient Selection on Program Generation

Hiroto Watanabe*, Shinsuke Matsumoto*, Yoshiki Higo*, Shinji Kusumoto*,
Toshiyuki Kurabayashi†, Hiroyuki Kirinuki†, and Haruto Tanno†
*Graduate School of Information Science and Technology, Osaka University, Japan
† Nippon Telegraph and Telephone Corporation, Japan
{h-watanb, shinsuke, higo, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—Automated program generation (APG) is a concept of automatically making a computer program. Toward this goal, transferring automated program repair (APR) to APG can be considered. APR modifies the buggy input source code to pass all test cases. APG regards empty source code as initially failing all test cases, i.e., containing multiple bugs. Search-based APR repeatedly generates program variants and evaluates them. Many traditional APR systems evaluate the fitness of variants based on the number of passing test cases. However, when source code contains multiple bugs, this fitness function lacks the expressive power of variants. In this paper, we propose the application of a multi-objective genetic algorithm to APR in order to improve efficiency. We also propose a new crossover method that combines two variants with complementary test results, taking advantage of the high expressive power of multi-objective genetic algorithms for evaluation. We tested the effectiveness of the proposed method on competitive programming tasks. The obtained results showed significant differences in the number of successful trials and the required generation time.**

***Index Terms*—Automated Program Generation, Automated Program Repair, Multi-Objective Genetic Algorithm**

I. Introduction

Automated program generation (APG) is a promising and exciting concept, which reduces the programming burden placed on developers. One solution to achieve APG is transferring automated program repair (APR) [1], [2] to APG. In APR, buggy source code, which fails one or more test cases, is repaired to pass all test cases using a metaheuristic search algorithm, such as a genetic algorithm (GA). APR focuses only on repair, which means that the repaired source code is considered to be almost implemented. In contrast to repairing, APG tries generating functionally competent source code from empty source code that initially fails all test cases.

Although numerous studies have been conducted on APR in the past decade [3], there remain many issues in terms of both practical and theoretical aspects. Specific issues include the following: requiring a large amount of computational power for fixing bugs [4], overfitting to test cases [5], generating incorrect but plausible patches [6], difficulty fixing multiple bugs [7]. APG regards the initial source code as empty, which means the code contains a number of bugs. From the perspective of transferring APR to APG, it is essential to solve the problem of fixing multiple bugs.

This study focuses on the selection phase of GA-based APR in order to address the issue of multiple bugs. Standard GA-based APR [1], [2], [8] repeats the process of modifying the source code (i.e., generation of variants) and evaluating it by running test cases (i.e., validation of variants). In the validation, fitness is calculated based on the degree to which the generated variants have improved for bug fixing. Variants with high fitness are selected for the next generation in a biological evolutionary process.

One problem with the selection is in determining a fitness function. Many APR techniques use the number of passing test cases as the fitness function [3]. This function is suitable when repairing a single bug. However, when repairing multiple bugs, it causes a problem due to insufficient representation. Let us consider a case where a variant passes two of three test cases. We denote this variant as `xx-`, meaning that it passes the first and second test cases. Here, two variants `x--` and `--x` are regarded as being the same by the traditional scalar fitness function because the number of passing test cases for each variant is one. However, `--x` should be selected preferentially in terms of providing complementary modifying information to `xx-`.

In this paper, we propose applying a multi-objective GA (MOGA) to APR in order to improve the efficiency of APG. The MOGA evaluates variants using multiple fitness functions, which enables the evaluation of variants with higher expressive power compared to a single scalar. The use of multiple fitness functions can also be expected to avoid local optima [9], so it has a high affinity with APG, which has a huge search space. We also propose a method that selects two variants with complementary test results and a new crossover method that combines them. In the experiment, we compare the efficiency of the proposed method with that of the existing APR tool using 80 competitive programming tasks. We find that the proposed method significantly improves both the number of successful trials and the generation time required compared to the existing methods.

II. Preliminaries

We first provide an overview of APG, which GA-based APR is transferred to. Additionally, we highlight a challenge that existing methods face with concrete examples.

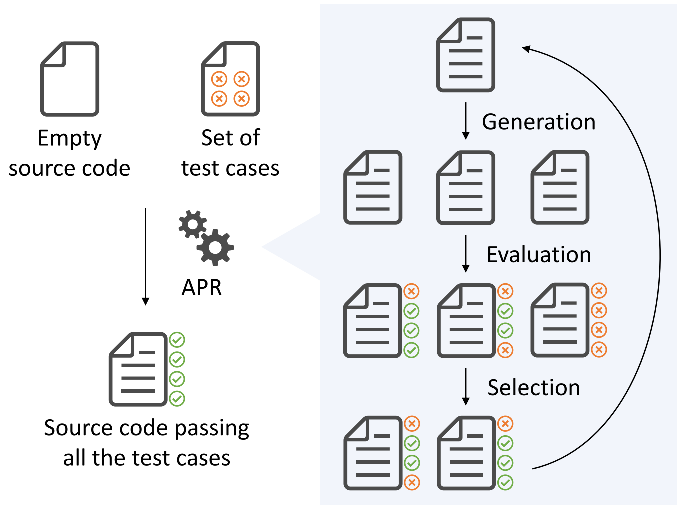


Fig. 1. Overview of transferring GA-based APR to APG

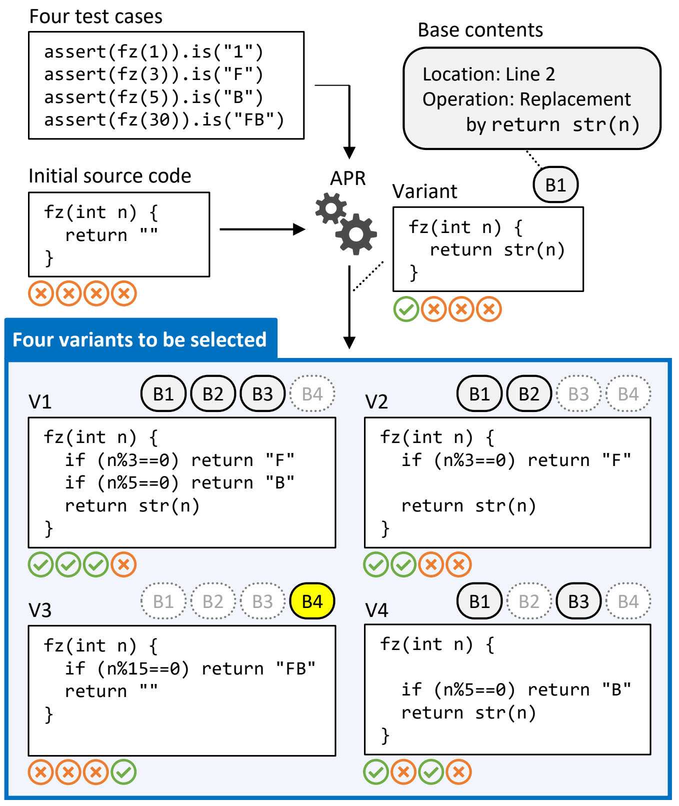
A. Overview of Automated Program Generation

Fig. 1 shows an overview of search-based APG. The input is empty source code and test cases, and the output is source code that passes all test cases. Search-based APG consists of three iterative processes: generation, evaluation, and selection of variants. In the following, we explain these three processes based on the GA used by GenProg [1], which was a breakthrough in the field of APR.

First, a variant is generated from the input source code using genetic operators. GenProg has two genetic operators: *mutation* and *crossover*. Mutation generates a new variant by insertion, deletion, or replacement of any particular statement. Crossover generates new variants by reusing the modification history (i.e., *gene*) of two variants. Next, for each generated variant, we evaluate how close it is to our goal of fixing the bug. Fitness is calculated from source lines of code of the variant, the length of the gene, or the results of compiling the variant and executing the test cases. GenProg calculates fitness from the number of passing test cases. Based on the fitness, variants to be saved are selected.

B. Challenge of Existing Methods

Next, we explain a challenge that existing methods face. Fig. 2 shows four specific test cases based on the FizzBuzz and examples of variants. In this figure, four test cases and the initial source code, which always returns an empty string, are given as input to APR. As shown in the upper right of Fig. 2, each variant has a single gene consisting of a set of *bases*. Bases and genes can be designed in various ways, but here bases are composed of two elements: position and operation. Let us take the variant in the upper right of Fig. 2 as an example. This variant has a gene that consists of a single base, B1. The position of B1 is in the second line, `return ""`, and the operation of B1 is a replacement by `return str(n)`. Applying this gene, the variant has evolved from failing all test cases to passing the first one.



```
assert(fz(1)).is("1")
assert(fz(3)).is("F")
assert(fz(5)).is("B")
assert(fz(30)).is("FB")
```

Location: Line 2
Operation: Replacement by `return str(n)`

```
fz(int n) {
  return ""
}
```

```
fz(int n) {
  return str(n)
}
```

```
fz(int n) {
  if (n%3==0) return "F"
  if (n%5==0) return "B"
  return str(n)
}
```

```
fz(int n) {
  if (n%3==0) return "F"

  return str(n)
}
```

```
fz(int n) {
  if (n%15==0) return "FB"
  return ""
}
```

```
fz(int n) {

  if (n%5==0) return "B"
  return str(n)
}
```

Fig. 2. Challenge in selection phase

Let us assume that four variants, V1-V4, have been generated. The top-left variant, V1, is the best because it passes the most test cases. Therefore, V1 is used for the next generation. In search problems such as APR, it is common to select many variants instead of a single variant. This strategy aims to ensure genetic diversity and to avoid local optima.

When the number of passing test cases is used as a fitness measure, V2 and V4, which both pass two test cases, are selected after V1. However, when we check the passing test cases and the bases of the variants, we find that these two variants are a complete subset of V1 in terms of bases and the results from running test cases, so the selection is inefficient. On the other hand, although the bottom-left variant, V3, passes the least number of test cases, it passes the fourth test, which the other variants fail. Additionally, V3 has a special gene, B4, that other variants do not have, and it realizes the process of division by 15 in the FizzBuzz. Thus, in variant evaluation, the number of passing test cases is insufficient in terms of its expressive power, and it is necessary to compare the success or failure of each test case for each variant.

III. The Proposed Method

In order to improve the efficiency of APG, we propose two methods. The first involves applying a MOGA to APR. The other involves combining two variants that have complementary test case results, using the high expressiveness of evaluation by the MOGA.

A. *Multi-Objective Genetic Algorithm*

The MOGA has multiple fitness functions to evaluate variants. In the following, we explain the design of these functions and how to use them to select variants.

1) Fitness Functions: This section describes the fitness functions that evaluate variants. In order to solve the selection problem described in Section II-B, the result of each test case is used as an independent fitness function. That is, if there are M test cases, the number of fitness functions is M. The ith fitness function $f_i(v)$ returns 1 if a variant v passes the ith test case, and 0 if it fails. A variant whose M fitness function values are all 1 passes all test cases.

2) Definition of a Relation between Variants: For two variants, v_a and v_b, we say that v_b *dominates* v_a if $f_i(v_a) < f_i(v_b)$ $(\forall i = 1, \ldots, M)$. At this time, a partial relation $\prec$ is defined as $v_a \prec v_b$. A set of variants forms a lattice under the relation $\prec$.

3) Selection: Variants to be left for the next generation are determined based on the fitness of each variant. Specifically, variants are ranked using the Pareto ranking method [10]. In this method, the rank of a variant v is $1+$ the number of variants that dominate v in the generation to which v belongs. Between two variants with different ranks, we select the variant with the better (i.e., lower) rank.

Fig. 3 shows a Hasse diagram of the relation between all possible variants when there are four test cases. In this figure, a line is drawn upward from v_a to v_b when $v_a \prec v_b$ is satisfied and there is no v_c such that $v_a \prec v_c \prec v_b$. The top variant, which dominates all other variants, is the target variant that passes all test cases. When there are all the variants shown in Fig. 3, the ranks of each variant are: 1 for the top variant, 2 for the second four variants, 4 for the third six variants, 8 for the fourth four variants, and 16 for the bottom variant.

The ranks of the variants shown in Fig. 2 are 1 for V1 and V3, and 2 for V2 and V4. This is because V1 and V3 have no variants that dominate them in Fig. 3, but V2 and V4 have V1 above them. Using this rank, V3 can be selected in preference to V2 and V4.

B. *Crossover*

In addition to the application of the MOGA described in Section III-A, this study improves the crossover, which is one of the operations to modify the source code. In this section, we explain our proposed new crossover method, *cascade crossover*, which selectively combines two variants with complementary test results.

1) Cascade Crossover: Cascade crossover generates two variants by joining all of the bases of two variants. If two variants, v_a and v_b, are selected as parents, cascade crossover generates a new variant according to the following processes.

1) Joins the genes in the order v_a, v_b and removes duplicated bases.
2) Joins the genes in the order v_b, v_a and removes duplicated bases.
3) Generates two variants with these genes.

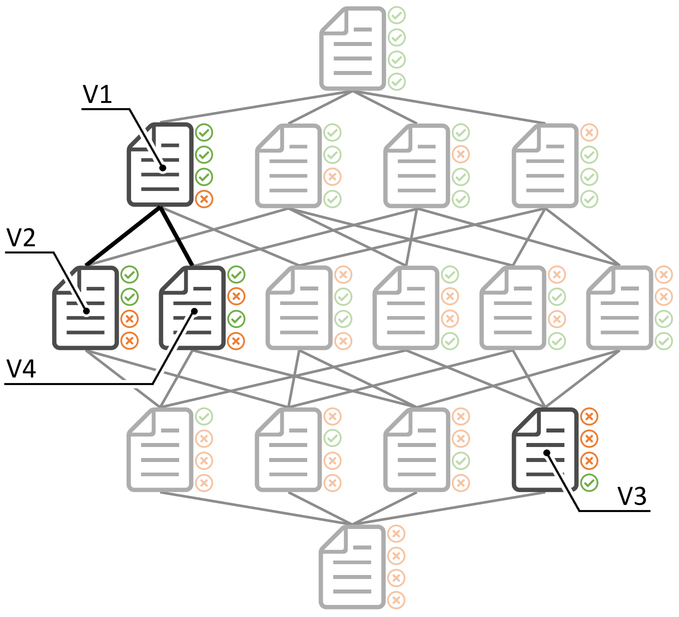


Fig. 3. Hasse diagram of all variants with four test cases, and the relation between V1-V4

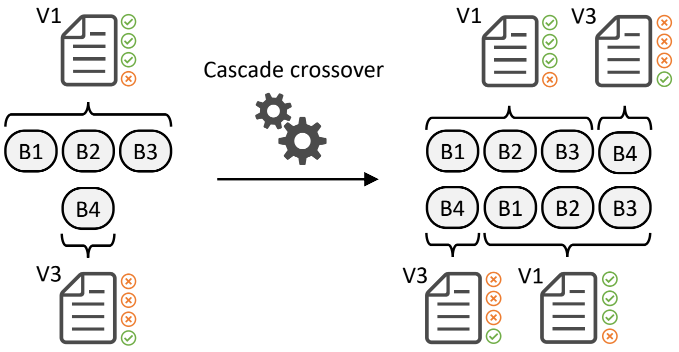


Fig. 4. Genes of V1 and V3 combined by cascade crossover

Fig. 4 shows the variants generated by cascade crossover of V1 and V3 in Fig. 2. The duplicated base, B2, is removed during the crossover processes.

The reason for removing duplicate bases is that they are likely to have a negative effect on the variant. As an example, let us consider a base that inserts a variable declaration. If this base is duplicated, two variables with the same identifier will be declared, and the variant will fail to compile.

2) Selection of Variants to Crossover: Among the variants with rank 1, the cascade crossover is performed on two pairs of variants whose fitness do not match. Selecting variants in this way aims to produce variants that pass all of their parents' passing tests. In the case of the four variants shown in Fig. 2, the variants of rank 1 are V1 and V3, so cascade crossover is performed only on these.

3) Validation of Variants Generated by Crossover: When cascade crossover generates a variant, v_c, by combining v_a and v_b, v_c is selected for the next generation only if v_c dominates v_a and v_b. This is because, as mentioned in the previous section, the goal of cascade crossover is to generate variants that pass all of the tests that the parents passed. In the case of combining V1 and V3, the generated variants are eligible for selection in the next generation only if they pass all four test cases.

IV. Experiment

A. Overview of Experiment

To confirm its effectiveness, we implemented the proposed method based on an existing APR tool, kGenProg [8], which is a Java implementation of GenProg. The purpose of this experiment is to investigate how the proposed method changes the efficiency of APG. We compared our tool with kGenProg before the extension. The evaluation metric consist of the number of successes, which means generating a variant that passes all of the input test cases, and the time required for APG.

B. Experimental Settings

A list of the experimental settings is shown in Table I. We used eighty 100-point tasks from ABC101 to ABC180 of the past AtCoder Beginner Contest (ABC), held at AtCoder[1]. Since kGenProg and our tool use GA, which has randomness, we set 20 random seeds (from 1 to 20) and tried generating source code.

Our tool and kGenProg both require source code to generate variants. Therefore, we obtained source code fragments from all correct answers of the 80 tasks used in the experiment and flattened their nested structures. In the flattening process, each statement in the blocks (e.g., `if` and `for`) was moved out of the blocks. Since the reused code includes all statements that make up the correct code, it is theoretically possible to generate the correct code if sufficient time is spent on this task. In this way, we were able to check whether it is possible to generate source code only by reusing small independent code snippets. We used all of the test cases published by AtCoder as input for both tools. The default values of kGenProg were used for the other parameters necessary for the GA.

C. Results

1) Number of Successes: We call a task that has at least one successful trial out of 20 trials a *successful task*. The number of successful tasks generated by both tools was 62, only one by kGenProg was 4, only one by our tool was 4, and the number of tasks unsuccessfully generated by both tools was 10.

Comparing the number of successfully generated trials for each task, our tool outperformed kGenProg in 35 tasks and kGenProg outperformed our tool in 20 tasks. Out of 1,600 trials, there were 730 successful trials for kGenProg and 784 for our tool. We conducted a Wilcoxon signed-rank test for the number of successful trials and found a significant difference between kGenProg and our tool ($p = 4.46 \times 10^{-3}$).

2) Source Code Generation Time: Fig. 5 shows the average generation time for each of the 62 both successful tasks. The horizontal axis represents the task name, and the vertical axis represents the average time required for successful generation. We sorted the tasks on the horizontal axis by the time taken to execute kGenProg in descending order. There were 43 tasks that succeeded in less time with our tool, and 19 tasks with kGenProg. We conducted a Wilcoxon signed-rank test for the average time and found a significant difference between kGenProg and our tool ($p = 2.48 \times 10^{-3}$).

[1]https://atcoder.jp/

TABLE I
Experimental Settings

Item	Value
Subject of experiment	ABC101∼ABC180 100-point task
Number of tasks	80
Random seed	1∼20 (20 trails)
Time limit	1 hour per trial
Reuse code	Correct code snippets for all subjects
Generation limit	Unlimited
CPU mem	Xeon E5-2630 2 CPUs 16 GB mem

V. Discussion

It can be concluded that the proposed method can generate more correct source code in a shorter time than the conventional method, and that the proposed method is more efficient than the conventional method.

The Wilcoxon signed-rank test showed significant differences in the number of successful trials and the program generation time. It can be concluded that the proposed method generates source code faster than the conventional method. Therefore, the proposed method is considered to improve the efficiency of source code generation.

From Fig. 5, it can be seen that the shorter the generation time is in the conventional method, the smaller the efficiency improvement obtained by the proposed method. We now consider the reason for this. As a concrete example of a task whose generation time is short in the conventional method, we choose ABC134. In ABC134, the input is a and the output is $3a^2$. In this task, only one piece of code is required (i.e., `return 3*a*a`), and if we can insert that code, we can generate a variant that passes all input test cases. In such a simple task, generated variants either pass or fail all the test cases, so high expressiveness of the fitness is not necessary. On the other hand, ABC123, which has the largest difference in generation time, requires a branch (e.g., `if`) in the correct code, which causes the problem described in Section II-B. From the above, we can see that the proposed method solves the problem associated with the selection phase.

VI. Related Works

Many APR tools have been proposed. ARJA [2] is a search-based APR tool with MOGA. ARJA has two fitness functions: the number of passing test cases and the patch size. ARJA aims to improve readability of generated patches by introducing patch size as one of the multiple fitness functions.

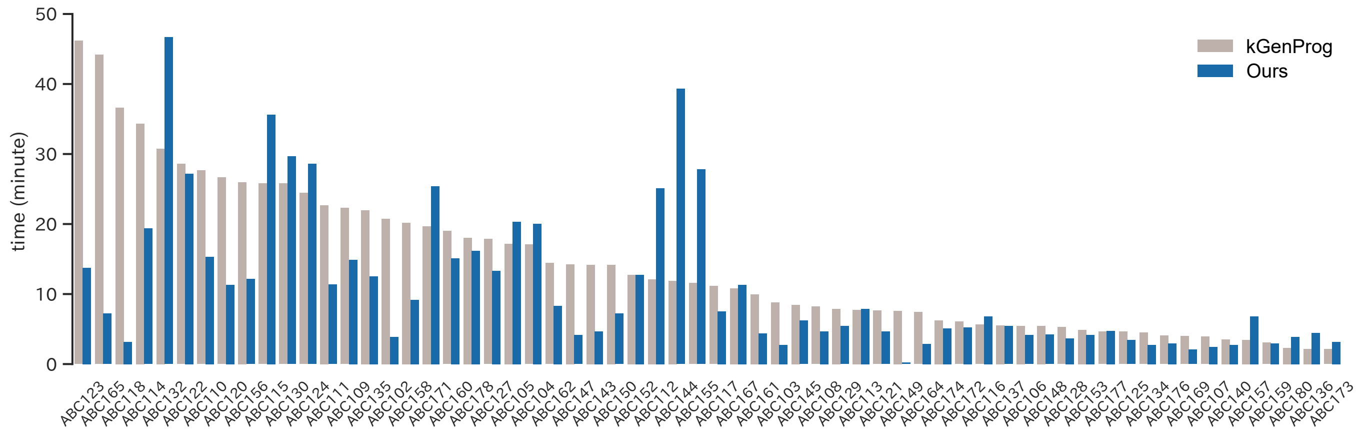


Fig. 5. Average generation time for each task

Similar to our approach, ARJAe [11] and 2Phase [12] have fine-grained fitness functions to improve the selection phase. ARJAe is an extension of ARJA. In addition to the above two fitness functions, ARJAe takes assertion distances into consideration. In other words, ARJAe calculates the distances from the difference between expected and actual assertion values. A similar idea is introduced into 2Phase. 2Phase has a scalar fitness function calculated from the number of passing test cases and the assertion distances.

The main difference between our method and these three tools is the design of the fitness functions. Our design shown in Section III-A1 enables us to select different variants which complement each other. On the other hand, these three tools use the number of passing test cases as the fitness function. As we described in Section II-B, this fitness function lacks the expressive power of variants. This causes these three tools to suffer from the challenges shown in Fig. 2.

VII. Conclusions and Future Work

In this paper, we proposed a method for APG by applying a MOGA, in which each test result is an independent fitness function. The effectiveness of the proposed method was confirmed through an experiment using 80 competitive programming tasks.

In future work, we intend to include a detailed analysis of the experimental results. We also intend to analyze the effect of each subject on the proposed method, e.g., ABC144, where the generation time for the proposed method deteriorated significantly. Additionally, the degree to which the MOGA proposed in this paper contributed to the efficiency improvement of the selection method and cascade crossover is unclear. Although the MOGA enables the avoidance of local optima, we have not confirmed whether this effect is obtained by our method, so a detailed analysis is needed. In terms of improving the method, we are considering incorporating metrics such as lines of code or test time. In addition, to improve the cascade crossover, instead of using all the bases, we could use a dynamic source code analysis to attempt to select only the useful bases that can pass the test. In our experiments, we used the simplest tasks of AtCoder Beginner Contest, but application to more complex tasks is an issue for future investigation.

Acknowledgment

This research was partially supported by JSPS KAKENHI Japan (Grant Numbers: JP20H04166, JP21H04877).

References

[1] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A Generic Method for Automatic Software Repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[2] Y. Yuan and W. Banzhaf, "ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming," *IEEE Transactions on Software Engineering*, vol. 46, no. 10, pp. 1040–1067, 2020.

[3] L. Gazzola, D. Micucci, and L. Mariani, "Automatic Software Repair: A Survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2017.

[4] F. Long and M. Rinard, "An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems," in *Proc. International Conference on Software Engineering*, 2016, pp. 702–713.

[5] 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. Joint Meeting on Foundations of Software Engineering*, 2015, pp. 532–543.

[6] Z. Qi, F. Long, S. Achour, and M. Rinard, "An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems," in *Proc. International Symposium on Software Testing and Analysis*, 2015, pp. 24–36.

[7] S. Saha, R. K. Saha, and M. R. Prasad, "Harnessing Evolution for Multi-Hunk Program Repair," in *Proc. International Conference on Software Engineering*, 2019, pp. 13–24.

[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 *Proc. Asia-Pacific Software Engineering Conference*, 2018, pp. 697–698.

[9] J. D. Knowles, R. A. Watson, and D. W. Corne, "Reducing Local Optima in Single-Objective Problems by Multi-objectivization," in *Proc. International Conference on Evolutionary Multi-Criterion Optimization*, 2001, pp. 269–283.

[10] C. M. Fonseca and P. J. Fleming, "Genetic Algorithms for Multiobjective Optimization: FormulationDiscussion and Generalization," in *Proc. International Conference on Genetic Algorithms*, 1993, pp. 416–423.

[11] Y. Yuan and W. Banzhaf, "Toward Better Evolutionary Program Repair: An Integrated Approach," *ACM Transactions on Software Engineering and Methodology*, vol. 29, no. 1, pp. 1–53, 2020.

[12] Z. Bian, A. Blot, and J. Petke, "Refining Fitness Functions for Search-Based Program Repair," in *Proc. International Conference on Software Engineering Workshops*, 2021, pp. 1–8.