Toward Developer-like Automated Program Repair —Modification Comparisons between GenProg and Developers—

Hiroki Nakajima, Yoshiki Higo, Haruki Yokoyama, and Shinji Kusumoto
Graduate School of Information Science and Technology
Osaka University
1-5 Yamadaoka, Suita, Osaka, Japan
Email: {h-nakajm,higo,y-haruki,kusumoto}@ist.osaka-u.ac.jp

***Abstract*—Automated program repair is a way to reduce costs on program debugging to a large extent. Repair techniques using genetic programming have been attracting much attention. They were applied to actual software systems and they were able to fix several dozen of actual faults. However, programs generated by such techniques often include some source code changes not related to fixing a given fault even if they pass all given test cases. Furthermore, some researchers found that such techniques occasionally induce new faults which are not covered by existing test cases. The reason why those problems arise is that such techniques consider only given test cases. On the other hand, developers consider program behaviors not covered by test cases. Thus, those problems arise less frequently in programs modified by developers. Consequently, the authors suppose that if we make automated program modifications close to developers' ones, we may be able to relieve those problems. At this moment, there is no research study investigating differences between automated modifications and developers' ones. In this paper, we compare GenProg's modifications with developers' ones for the same faults. As a result, we found that developers tend to (1) change more different functions, (2) change control flows in source code, and (3) add/delete more code lines.**

***Keywords*-Automated program repair; Source code analysis; Genetic programming;**

I. Introduction

Debugging is an inevitable activity in software development. Debugging occasionally requires a large amount of human resources. There is a report that debugging took more than half of total software development cost [1]. Besides, in the United States, 300 billion dollars are spent for debugging every year [2]. Consequently, a variety of research on debugging support has been conducted.

Automatizing program debugging is a way to reduce its costs. There are many studies on fault localization [3], [4]. However, to promote more automated debugging, source code modification need to be conducted automatically. Recently, to conduct debugging in a fully-automatic way, automated program repair techniques have been attracting much attention. At present, there are many methodologies and tools to conduct automated program repair. Those methodologies can be classified into two types, search-based and semantic-based approaches. The search-based approach (also known as generate-and-validate approach) searches within a defined search space to generate a repair candidate and validate it with a given set of test cases. GenProg [5], PAR [6], SPR [7], HDRepair [8], and Prophet [9] are classified into the search-based approach. The semantic-based approach synthesizes a repair by leveraging semantic information of the buggy program. For example, symbolic execution and constraint solving techniques are usually used in the semantic-based approach. Angelix [10], DirectFix [11], SemFix [12], and Nopol [13] are classified into the semantic-based approach.

GenProg conducts automated program repair by using genetic programming [5]. GenProg takes a faulty program and test cases as its input. Its output is a modified program, which passes all the given test cases. GenProg firstly localizes the faults by using each of the given test cases. Then, GenProg conducts source code changes in the localized code fragments.

In GenProg, there are three kinds of operations to generate modified programs.

[Insertion] inserting a program statement to the next line of the localized code line.

[Deletion] deleting a statement in the localized code line.

[Replace] conducting both insertion and deletion.

Programs generated by the above operations are validated with the given test cases. If a generated program passes all the test cases, it is output as a modified program. If there are no programs passing all the test cases, one of the above operations is applied to each generated program to generate programs of the next generation. This procedure is repeated until a generated program passes all the test cases.

GenProg's output is a program passing all the test cases. However, GenProg does not guarantee correct behaviors that are not covered by the given test cases. In other words, an output program may include new faults. There is a report that there were some cases where GenProg's output lacked required functionality or included new faults [6], [14]. On the other hand, developers use a broader and deeper knowledge of what the program is intended to be. They modify programs with consideration of program behaviors not covered by test cases [15]. Thus, programs modified by

developers can be considered less including the above issues than programs modified by GenProg. The authors consider one of the next steps of automated program repair is making GenProg's modifications close to developers' ones as much as possible. To do this, we need to know how GenProg's modifications are different from developers' ones.

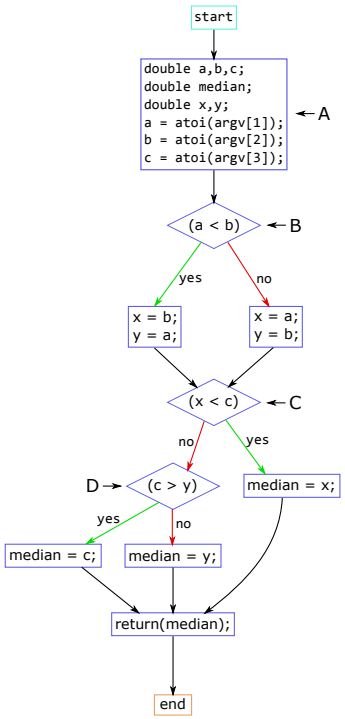
In this research, we conducted an exploratory study to investigate where and how GenProg and developers had changed given faulty programs. Our main findings are as follows.

- Developers tend to change more different functions than GenProg.
- Developers tend to change programs' structure more than GenProg.
- Developers tend to add/delete more program statements more than GenProg.

The remainder of this paper is organized as follows: Section II describes our research purpose; in Section III, we introduce the control flow graph and explain the procedures of our experiments; Section IV shows the experimental results and Section V discusses future research directions of automatic program repair; lastly, Section VII concludes this paper.

```
1 int main(
2     int argc,
3     char *argv[]){
4
5   double a, b, c;
6   double median;
7   double x, y;
8
9   a = atoi(argv[1]);
10  b = atoi(argv[2]);
11  c = atoi(argv[3]);
12
13  if (a < b){
14    x = b;
15    y = a;
16  }
17  else {
18    x = a;
19    y = b;
20  }
21
22  if (x < c)
23    median = x;
24  else if (c > y)
25    median = c;
26  else
27    median = y;
28
29  return (median);
```

(a) Source code



(b) CFG

Figure 1: An example of CFG

II. Research Purpose

GenProg was applied to eight open source software and 55 out of 105 faults were fixed [16]. The results show GenProg's capability to fix faults is promising. GenProg regards a generated program passing all the given tests as a modified program. GenProg does not consider readability, maintainability or other source code features to generate programs. GenProg's modified programs occasionally include some changes not related to fixing given faults [6]. Besides, there were some cases where GenProg's modified programs included the following issues [14].

- Modified programs lacked required functionality.
- Modified programs included new faults.

Contrary to GenProg, developers consider program behaviors that are not covered by its existing test cases when they modify programs [15]. Thus, programs modified by developers are considered less likely to include the above issues than programs modified by GenProg.

In this research, we compare GenProg's modifications with developers' ones. The findings of this research will be useful to make GenProg's modifications close to developers' ones.

In order to realize more developer-like automated program repair, we investigate the following research questions.

[RQ1] do GenProg and developers modify the same functions for the same faults?

[RQ2] do GenProg and developers modify programs in the same ways for the same faults?

Qi et al. reported that GenProg and other tools tend to delete existing program statements to pass given test cases [17]. However, they did not directly compare the degree of deletions likelihood of the tools with developers, which we do in this research. Besides, we conducted direct comparison on several kinds of program elements such as if-statement and while-statement.

III. Experimental Design

Herein, we firstly describe control flow graph, which we use to compare GenProg's and developers' modifications. Then, we explain the experimental procedure.

A. Control Flow Graph

Control flow graph (hereafter, CFG) is a graph representing control flows in program source code. Every node in a CFG is a code fragment including neither bifurcations nor confluences. Control flows between such code fragments are depicted as direct edges. A CFG is generated for each function in a given program. If loops or branches in a given function are changed, the structure of its CFG is also changed.

Figure 1 shows an example of source code and its CFG. The source code has functionality to identify the median value from numerical values specified by the parameter. Nodes B, C, and D are bifurcation nodes. They correspond to if-statements in the source code. Each of the bifurcation nodes is a 1-line code fragment while node A is a 6-line

code fragment. There are 10 code fragments in this source code.

In this research, we manually investigate how GenProg and developers modify programs. We leverage UNIX diff and visual presentations of their CFGs so as to easily and correctly understand how programs were modified.

B. Target

Our experimental target is ManyBugs benchmarks [18], which includes nine open source software written in C. Table I shows an outline of the target software. The column *#Faults* shows two numbers. The number outside the parentheses means the number of all the faults included in the software and the one inside the parentheses means the number of the faults that GenProg was able to fix. In this experiment, we investigate the faults that GenProg was able to fix, which are totally 83.

ManyBugs benchmarks also include developers' modified program for each fault. The developers' modified programs include not only modifications to fix faults but also feature additions. The authors of literature [18] checked the developers' modified programs one by one. The followings are the four most common defect categories:

- instances of incorrect behavior or incorrect output,
- segmentation faults,
- fatal errors (non-segmentation fault crashes of otherwise unspecified type), and
- instances of feature additions.

In this experiment, we leverage the developers' modified programs for the 83 faults that GenProg was able to fix. Before manually investigating programs for the 83 faults, we conduct a preprocessing on the faulty programs and the developers' modified programs because GenProg's modified programs do not include for-statements at all. If a faulty program includes for-statements, they are transformed to while-statements in GenProg's text→AST→text transformation even if GenProg does not apply modifications on them. Consequently, we use GenProg's component to apply the same transformations to the faulty programs and the developers' modified programs. After the preprocessing, all three kinds of programs do not include for-statements at all.

Table I: Target software

Software	kLOC	#Faults	Description
fbc	97	3 (1)	compiler
gmp	145	2 (0)	mathematic library
gzip	491	5 (1)	data compression
libtiff	77	24 (17)	graphics library
lighttpd	62	9 (4)	Web server
php	1,099	104 (51)	programming language
python	407	15 (2)	programming language
valgrind	793	15 (3)	debugging tool
wireshark	2,814	8 (4)	network analyzer
Sum	5,985	185 (83)	

C. Procedure

For each fault, we conduct investigations with the experimental procedure shown in Figure 2. The inputs are three kinds of programs, a program including a given fault, a program modified by a developer, and a program modified by GenProg. The procedure to investigate a given fault consists of the following four steps.

[STEP1] identifying changed code lines by using UNIX diff.
[STEP2] generating CFGs.
[STEP3] investigating which code fragments of the faulty program were changed to fix the given fault.
[STEP4] investigating how the faulty program was changed to fix the fault.

The remainder of this section describes the steps in detail.

STEP1 is an operation to identify which code lines were changed by a developer and GenProg. In this step, we use UNIX diff two times. The first time is to compare the faulty program and the program modified by a developer. The second time is to compare the faulty program and the program modified by GenProg. In this step, we also manually investigate which functions were modified.

STEP2 is a simple operation. We generate a CFG for each modified function in the faulty program, the program modified by developers and the program modified by GenProg. We used Understand[1] to generate CFGs.

In STEP3, we investigate which code fragments[2] were modified by developers and GenProg.

In STEP4, we investigate what kinds of modifications were conducted by developers and GenProg. In this investigation, modifications are divided into two categories. The first one is modifications that changed the structure of the CFG. The second one is ones that did not change.

Regarding the first category, changing the structure of the CFG means branches and/or loops in source code were changed by the modifications. We count the number of added and deleted if-statements for branch modifications. We also count the number of added and deleted while-statements for loop modifications. Please note that the programs do not include for-statements at all as described in Subsection III-B.

Regarding the second category, we count the number of CFG nodes whose code fragments were modified. We also count the number of added and deleted lines of code for each of the modified code fragments.

After we count all the above numbers, we compare developers' modifications and GenProg's ones with Mann-Whitney U-test to check whether those modifications have significant differences. We used 0.01 as significance level. In this statistical testing, null hypothesis and alternative hypothesis are as follows.

[1]https://scitools.com/

[2]In this experiment, the source code of every CFG node is regarded as a code fragment.

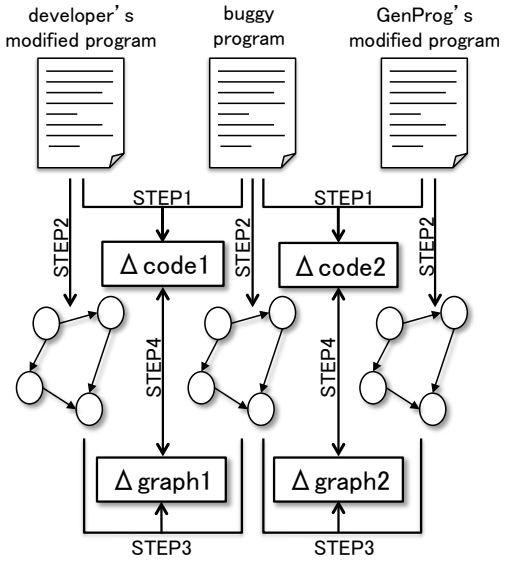


Figure 2: An overview of the experimental procedure

[Null hypothesis (H0)] there is no difference between programmers' modifications and GenProg's ones.

[Alternative hypothesis (H1)] programmers' modifications are different from GenProg's ones.

IV. Experimental Results

We investigated the 83 faults shown in Table I with the procedure described in Subsection III-C. Developers and GenProg modified 143 and 111 functions, respectively. Only 32 out of the functions were modified by both developers and GenProg.

Table II shows a summary of the investigation. As shown in this table, developers tend to change a greater amount of source code to fix faults than GenProg. Developers also tend to add/delete branches and loops in source code.

The remainder of this section describe the results of each evaluation item.

A. The number of added/deleted CFG nodes

Figure 3 is a boxplot showing how many nodes of a CFG (code fragments in the source code) were added or deleted by developers and GenProg. Developers added or deleted nodes on CFGs of 105 out of the 143 modified functions. On the other hand, GenProg added or deleted nodes on CFGs of 59 out of the 111 modified functions.

- Regarding the number of added nodes per CFG, the median value of developers is greater than GenProg's one. The p-value of U-test is 3.203×10^{-10}, which means H0 is rejected. Consequently, we cannot say that the two groups are the same.
- Regarding the number of deleted nodes per CFG, the p-value of U-test is 0.8823. Consequently, we can say that the two groups are not different significantly.

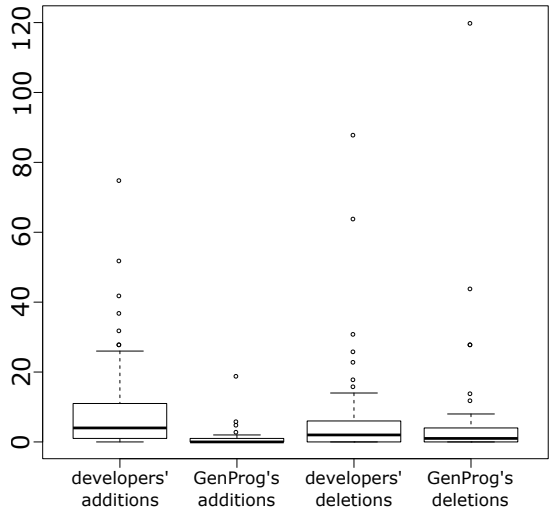


Figure 3: The number of added or deleted CFG nodes. Regarding developers, the number of data is 105, which is the number of functions where developers added or deleted code fragments. Regarding GenProg, the number of data is 59, which is the number of functions where GenProg added or deleted code fragments.

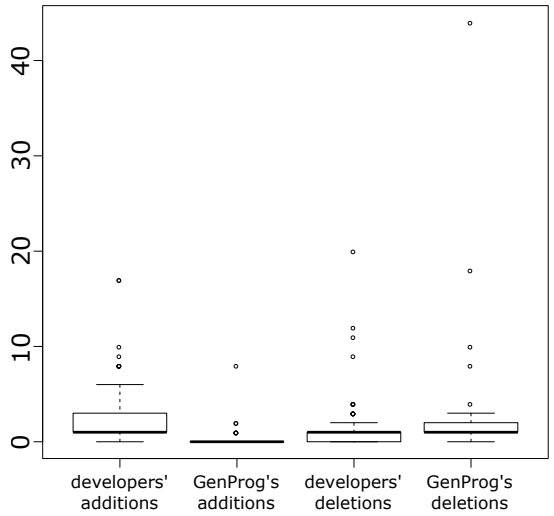


Figure 4: Added/deleted if-statements. The number of data of developers and GenProg is 90 and 33, respectively.

B. Added/Deleted if-statements

Figure 4 shows comparisons between developers and GenProg from the viewpoints of added/deleted if-statements. Developers added or deleted if-statements on 90 out of the 143 modified functions. On the other hand, GenProg did that on 33 out of the 111 modified functions.

- Regarding the number of added if-statements per function, developers' median value is greater than GenProg's one. Furthermore, the p-value of U-test is 3.742

Table II: Summary of investigation results

(a) Modifications adding/deleting CFG nodes

Evaluation item	p-value	Results
# Structure-changed CFGs	not statistical testing	Developers > GenProg
# Added Nodes	3.203×10^{-10}	Developers > GenProg
# Deleted Nodes	0.8823	Developers ≒ GenProg
# Added if-statements	3.742×10^{-7}	Developers > GenProg
# Deleted if-statements	0.06233	Developers ≒ GenProg
# Added while-statements	1.871×10^{-3}	Developers > GenProg
# Deleted while-statements	0.06233	Developers ≒ GenProg
# Added goto-statements	1.600×10^{-3}	Developers > GenProg
# Deleted goto-statements	0.3566	Developers ≒ GenProg
# added switch-statements	no value†	Developers > GenProg
# deleted switch-statements	no value†	Developers > GenProg

†GenProg neither add nor delete switch-statements at all.

(b) Modifications in changed CFG nodes

Evaluation item	p-value	Results
# Changed Nodes	not statistical testing	Developers > GenProg
# Added code lines	1.854×10^{-10}	Developers > GenProg
# Deleted code lines	1.373×10^{-4}	Developers > GenProg

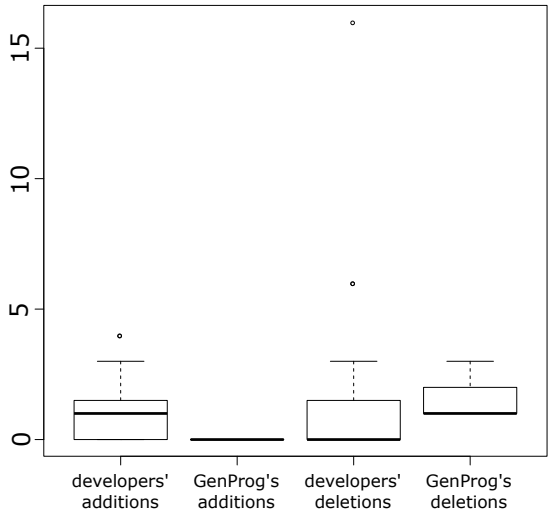


Figure 5: Added/deleted while-statements. The number of data of developers and GenProg is 23 and 9, respectively.

$\times 10^{-7}$. That is, we cannot say that there is no significant difference between the two groups.

- Regarding the number of deleted if-statements per function, the median values of the two groups are the same. The p-value of U-test is 0.06233, which means that null hypothesis is not rejected. Thus, we can say that the two groups are not different significantly.

C. Added/Deleted while-statements

In Figure 5, the number of added/deleted while-statements shown as boxplot representation. The number of functions where while-statements were added or deleted is smaller than ones where if-statements were added or deleted. More concretely, developer added/deleted while-statements on 23 out of the 143 modified functions and GenProg did that on only 9 out of the 111 modified functions.

- Regarding the number of added while-statements per function, GenProg did not add while-statements to fix faults. The p-value of U-test is 1.871×10^{-3}. Consequently, we cannot say that the two groups are not significantly different.
- Regarding the number of deleted while-statements per function, GenProg's median value is greater than developers' one, but the p-value of U-test is 0.06233. Consequently, we can say that the two groups are not significantly different.

D. Added/Deleted goto-statements

Figure 6 shows the number of added/deleted goto-statements by developers' or GenProg's modifications. Surprisingly, there are many functions where goto-statements were added or deleted to fix faults. Developers added or deleted goto-statements on 40 out of the 143 modified functions. On the other hand, GenProg did that on 30 out of the 111 modified functions.

- Regarding the number of added goto-statements per function, the median values of the two groups are the same. However, the p-value of U-test is 1.600×10^{-3}, which means we reject null hypothesis. Consequently, we cannot say that the two groups are not significantly different from each other.
- Regarding the number of deleted goto-statements for function, developers' median value is greater than GenProg's one. However, null hypothesis is not rejected because the p-value of U-test is 0.3566. Consequently, we can say that the two groups are not significantly different from each other.

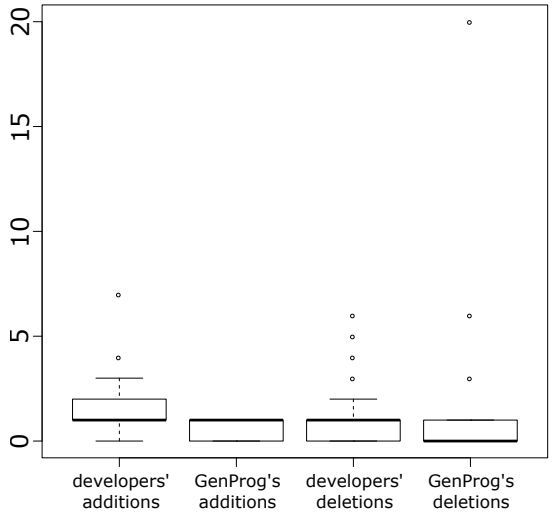


Figure 6: Added/deleted goto-statements. The number of data of developers and GenProg is 40 and 30, respectively.

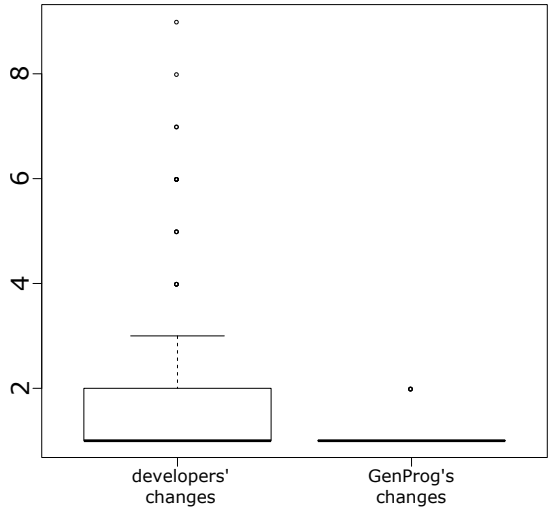


Figure 7: Changed CFG nodes. The number of data of developers and GenProg is 116 and 60, respectively.

E. Added/Deleted switch-statements

Developers added or deleted switch-statements on 3 out of the 143 modified functions. On the other hand, GenProg did not add or delete switch-statements for all the 111 modified functions.

F. The number of changed CFG nodes

Developers changed existing CFG nodes (code fragments) on 116 out of the 143 modified functions and GenProg did that on 60 out of the modified 111 functions. Figure 7 shows the number of changed nodes in developers' and GenProg's modifications on each of the 116 and 60 functions. GenProg changed only one or two code fragments in a function while developers occasionally changed three or more code fragments. The p-value of U-test on the two groups is 1.093×10^{-7}. Consequently, we cannot say that there is no significant difference between developers' and GenProg's modifications.

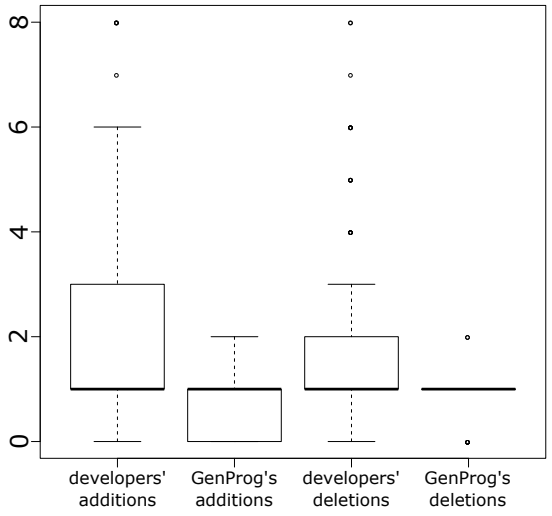


Figure 8: Added/deleted code lines. The number of data of developers and GenProg is 116 and 60, respectively.

G. Added/Deleted code lines

Figure 8 shows the number of added or deleted code lines in the changed nodes (code fragments). The numbers of the functions where their nodes were changed by developers' or GenProg's modifications are 116 out of 143 and 60 out of 111, respectively.

- Regarding the number of added code lines per function, the median value of the two groups are the same in spite that developers' graph is located the upper area than GenProg's one. The p-value of the two groups is 1.854×10^{-10}, and so the null hypothesis is rejected. Consequently, we cannot say that those two groups are not significantly different from each other.
- Regarding the number of deleted code lines per function, the median value of the two groups are also the same. The p-values of U-test is 1.373×10^{-4}. Consequently, we cannot say that those two groups are not significantly different from each other.

V. Discussion

We answer the two RQs based on the experimental results.

- Our answer to the RQ1 is **NO**. Developers and GenProg modified 143 and 111 functions, respectively. Only 32 out of them were commonly modified functions. That is, GenProg modified only 22.4% of the functions modified by developers. Besides, developers tend to modify more different functions than GenProg per fault.

- Our answer to the RQ2 is **NO**. Developers tend to add more branches/loops than GenProg. Besides, developers tend to add/delete more code lines to existing code fragments than GenProg.

From the above answers, the followings should be good research directions to make GenProg's modifications close to developers' ones.

Strategy1: modifying multiple functions: Developers tend to make multiple functions to fix a given fault. We can embed this strategy to GenProg. In GenProg's program generation, a single statement is different between n-th and $(n+1)$-th generation programs. For example, if a function is modified to generate a n-th program, modifying a statement in other functions to generate a $(n+1)$-th program is a straightforward way.

Strategy2: adding branches and loops: Developers tend to change control flows by adding branches and loops. We can also embed this strategy to GenProg. For example, in cases where insertion or replacement is performed to generate next generation programs, selecting if-statements or while-statements is a reasonable approach.

Strategy3: adding and deleting multiple code lines: Developers tend to add/delete more code lines. To imitate such modifications, adding/deleting multiple code lines in a single insertion/deletion operation should be a good approach. This strategy has already been realized as block insertion [15].

VI. Threats to Validity

This experiment includes a large amount of manual works. The authors may have done mistakes in the manual works. To avoid such mistakes as much as possible, the authors investigated each of the modified functions and their CFGs with fastidious care. The authors took approximately an hour to investigate every single function involved in the modifications on the 83 functions. The authors took a break every two- or three-function investigation to avoid a decrease in concentration.

In this experiment, the targets are nine software included in ManyBugs benchmarks. If we conduct the same experiments on other software developed by different developers, we may have obtained different results.

VII. Conclusions

In this paper, we reported comparison results between developers' modifications and GenProg's one. As a result, we obtained some findings: developers tend to modify more different functions than GenProg; developers tend to add more branches/loops than GenProg; developers tend to add/delete a greater amount of code lines than GenProg. In the future, we are going to propose and implement some strategies to improve automated program repair based on the findings.

VIII. Acknowledgments

This work was supported by JSPS KAKENHI Grant Numbers 25220003.

References

[1] J. Baker, "The gpl-violations.org project," http://goo.gl/yo8TkT, Feb. 2012, Last accessed 22 July 2016.

[2] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, "Reversible debugging software," Judge Business School, University of Cambridge, Tech. Rep., 2013.

[3] J. A. Jones and M. J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-localization Technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, 2005, pp. 273–282.

[4] X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang, "Taming Coincidental Correctness: Coverage Refinement with Context Patterns to Improve Fault Localization," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 45–55.

[5] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically Finding Patches Using Genetic Programming," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 364–374.

[6] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic Patch Generation Learned from Human-written Patches," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 802–811.

[7] F. Long and M. Rinard, "Staged Program Repair with Condition Synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 166–178.

[8] X.-B. D. Le, D. Lo, and C. L. Goues, "History Driven Program Repair," in *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*, 2016, pp. 213–224.

[9] F. Long and M. Rinard, "Automatic Patch Generation by Learning Correct Code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 298–312.

[10] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 691–701.

[11] ——, "DirectFix: Looking for Simple Program Repairs," in *Proceedings of the 37th International Conference on Software Engineering*, 2015, pp. 448–458.

[12] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program Repair via Semantic Analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 772–781.

[13] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. Lamelas, T. Durieux, D. L. Berre, and M. Monperrus, "Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs," *IEEE Transactions on Software Engineering (to be published)*.

[14] 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 *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 532–543.

[15] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun, "Repairing Programs with Semantic Code Search," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, 2015, pp. 295–306.

[16] 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.

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

[18] C. L. Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.