New Strategies for Selecting Reuse Candidates on Automated Program Repair

Akito Tanikado, Haruki Yokoyama, Masahiro Yamamoto, Soichi Sumi, Yoshiki Higo and Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University, Japan
Email: {a-tanikd, y-haruki, m-yamamt, s-sumi, higo, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—Automated program repair (in short, APR) is a truly desired technique because it can reduce debugging costs drastically. A well-known technique in APR is a reuse-based approach, which inserts existing program statements in a given program to suspicious code for an exposed bug. Some reports show the reuse-based approach was able to fix many bugs in open source software. However, the existing approach often takes very long time to fix bugs. Its main factor is that so many variant programs are generated by insertions and so many test cases are executed for the variant programs before a fixed program is generated. In order to shorten fixing time with the reuse-based approach, a fixed program must be generated much more efficiently. In this paper, we propose two strategies to generate a fixed program more efficiently. We also implement the two strategies and confirm that there are real bugs which the two strategies contribute to shortening fixing time.**

I. Introduction

Debugging is an inevitable task in software development, and it takes large costs. Recently, APR attracts much attention due to its latent capability to reduce debugging costs drastically. APR takes a buggy program and some test cases, and it outputs a fixed program, which passes all the given test cases. GenProg [1], a well-known approach in APR techniques, performs insertions and deletions to generate variant programs. In insertions, selected program statements existing in a given program are added to the suspicious code. Variant programs are evaluated with the given test cases. If a variant program passes all the test cases, it is outputted as a fixed program. Otherwise, next-generation variants are generated in the same way. Those operations are repeated until a given time limit.

In GenProg, program statements for insertions are randomly selected. Thus, if a given program is not small, so many program statements exist in the program, and inserting program statements contributing to fixing the bug is unacceptable odds. Consequently, strategies to select contributive program statements are required.

In this paper, we propose two strategies, similarity-order and freshness-order. We have implemented the two strategies and evaluated them with real bugs in open source software. As a result, we found that there are bugs that the two strategies were able to fix in a shorter time than the original GenProg.

II. Aproach

A. Similarity-order

In similarity-order, program statements included in code fragments with high similarity to the suspicious code are preferentially selected. With this strategy, the bugs caused by overlooking code clones can be efficiently repaired.

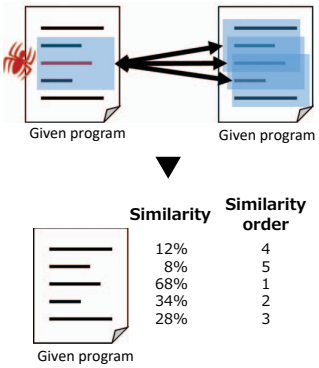

Fig. 1. Overview of similarity-order

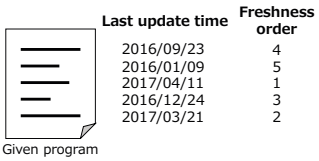

Fig. 2. Overview of freshness-order

Fig. 1 describes an overview of similarity-order. First, we consider a constant-sized region for each program statement in a given program. Then, we compute a similarity between each region and the suspicious code region. Finally, we select program statements in order of the similarity.

B. Freshness-order

As shown in Fig. 2, in freshness-order, program statements updated more recently are more preferentially selected. With this strategy, the bugs caused by failing to reflect changes involved with additions of new features can be efficiently fixed.

In freshness-order, development history of a buggy program, retrieved from version control system, is also given as input. For each program statement in a buggy program, last update time is taken from development history. We define the *freshness* of a program statement as how recently it is updated. Then, we select program statements in order of the *freshness*.

III. Experiment

To evaluate the usefulness of similarity-order and freshness-order, we implemented an APR tool with these strategies

0730-3157/17 $31.00 © 2017 IEEE
DOI 10.1109/COMPSAC.2017.175

by incorporating these strategies into Astor [2], an existing APR tool. We also evaluated these strategies by comparing results of using similarity-order and freshness-order with ones of original Astor (random), respectively. Evaluation items are as follows:

- number of repaired bugs, and
- repair time.

Experimental objects are 106 bugs[1] occurred in developing Apache Commons Math. These bugs are retrieved from Defects4J [3], a dataset of bugs.

A. Experimental Results

1) Number of Repaired Bugs: Random, similarity-order, and freshness-order fixed 27, 22, and 23 bugs, respectively. These results show neither similarity-order nor freshness-order fixed more bugs than random.

However, there are some bugs, which were not fixed by random but were by similarity-order and/or freshness-order (e.g. math56 and math64 in TABLE I). This means the number of repaired bugs can be increased by using similarity-order and/or freshness-order.

2) Repair Time: We conducted Wilcoxon signed-rank test for 18 bugs which had been fixed with all three strategies, to determine significant differences of repair time between similarity-order and random, freshness-order and random. The p-values for pairs of similarity-order and random, freshness-order and random were approximately 8.2×10^{-1} and 3.3×10^{-1}, respectively. This means there are no significant differences between similarity-order and random, nor freshness-order and random. Therefore, random is not faster than similarity-order nor freshness-order, and vice versa.

IV. Discussion

We found there were bugs the similarity-order and/or the freshness-order were able to fix more efficiently than the original Astor. On the other hand, the two strategies were not effective for other bugs. We conducted statistical tests and found there were no significant differences between the similarity-order/freshness-order and the original Astor. TABLE I shows actual time to fix bugs with the three strategies. We can see that

- there are some bugs a strategy was able to fix much more than the others, and that
- there are some bugs only a strategy was able to fix.

Thus, the three strategies have a complementary relationship. If we can select an appropriate strategy for a given bug, fixing time can be shortened drastically. However, at present, we have no standard to select an appropriate strategy and so it is impossible to know it before fixing bugs. A promising way to utilize our proposed strategies should be a round-robin execution, which performs either of the three strategies by rotation. Round-robin execution should be able to reduce fixing time averagely.

[1]The details of these bugs can be seen here: https://github.com/rjust/defects4j

TABLE I
REPAIR TIME (SEC) OF 30 BUGS REPAIRED BY AT LEAST ONE STRATEGY. "–" MEANS THE STRATEGY DID NOT REPAIR THE BUG.

Bug ID	Random	Freshness-order	Similarity-order
math2	198	11,183	8,110
math5	271	625	178
math7	5,798	–	–
math8	214	259	265
math12	3,330	8,521	1,075
math20	214	477	–
math22	406	115	112
math24	13,300	–	–
math28	164	175	199
math31	10,674	–	–
math40	8,009	922	1,089
math44	3,145	11,469	10,131
math49	473	1,657	277
math50	92	62	59
math53	31	4,903	24
math56	–	854	408
math60	5,588	10,744	512
math64	–	4,223	–
math70	175	82	60
math71	8,997	–	1,925
math73	24	283	111
math74	13,619	–	–
math77	–	11,256	–
math78	246	775	82
math80	157	841	–
math81	1,273	2,298	1,089
math82	496	653	514
math84	5,649	173	10,951
math85	26	–	42
math95	1,315	–	144

V. Conclusion

In this paper, we proposed two strategies, similarity-order and freshness-order, to shorten fixing time of APR. We have implemented the two strategies and evaluated them with real bugs. As a result, we found that the two strategies and the random-based selection, which is an existing strategy, have a complementary relationship. From this finding, we conclude that round-robin execution of the proposed strategies and the existing one can reduce fixing time averagely.

In the future, we are going to try to find characteristics of bugs to know an appropriate strategy before fixing bugs for more shortening fixing time. We are also going to implement a round-robin execution tool.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Number 25220003.

References

[1] 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 International Conference on Software Engineering (ICSE)*, 2012, pp. 3–13.

[2] M. Martinez and M. Monperrus, "Astor: a program repair library for java," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2016, pp. 441–444.

[3] 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 International Symposium on Software Testing and Analysis (ISSTA)*, 2014, pp. 437–440.