Toward Introducing Automated Program Repair Techniques to Industrial Software Development

Keigo Naitou*, Akito Tanikado*, Shinsuke Matsumoto*, Yoshiki Higo*, Shinji Kusumoto*,
Hiroyuki Kirinuki**, Toshiyuki Kurabayashi**, Haruto Tanno**
{k-naitou,a-tanikd,shinsuke,higo,kusumoto}@ist.osaka-u.ac.jp
*Osaka University, 1-5, Yamadaoka, Suita, Osaka, 565-0871, Japan
**Nippon Telegraph and Telephone Corporation, 2-13-34, Konan, Minato-ku, Tokyo, 108-8019, Japan

ABSTRACT

Automated program repair (in short, APR) has been attracting much attention. A variety of APR techniques have been proposed, and they have been evaluated with actual bugs in open source software. Currently, the authors are trying to introduce APR techniques to industrial software development (in short, ISD) to reduce development cost drastically. However, at this moment, there are no studies that report evaluations of APR techniques on ISD. In this paper, we report our ongoing application of APR techniques to ISD and discuss some barriers that we found on the application.

CCS CONCEPTS

• **Software and its engineering** → *Genetic programming*;

KEYWORDS

Automated program repair, Industrial application

ACM Reference Format:
Keigo Naitou*, Akito Tanikado*, Shinsuke Matsumoto*, Yoshiki Higo*, Shinji Kusumoto*, Hiroyuki Kirinuki**, Toshiyuki Kurabayashi**, Haruto Tanno**. 2018. Toward Introducing Automated Program Repair Techniques to Industrial Software Development. In *ICPC '18: ICPC '18: 26th IEEE/ACM International Confernece on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3196321.3196358

1 INTRODUCTION

Software developers spend 50% of their programming time to find and fix bugs [1]. Thus, debugging support is a hot topic in the field of software engineering. Debugging operation can be separated into two sub-operations: bug localization and bug modification. There are a variety of studies on bug localization [2–5]. Nowadays, automated program repair (in short, APR)—fully automated bug localization and modification—is attracting much attention.

GenProg is a breakthrough technique in APR [6]. GenProg takes a buggy program and a set of test cases and then it generates a modified program that passes all the given test cases. GenProg's bug modification is performed by deleting the localized code or inserting a code line to the localized code. A genetic algorithm is used in GenProg's bug modification process, which enables GenProg to generate a modified program even if a given buggy program includes multiple bugs and/or it requires many lines of code to remove bugs. Le Goues et al. applied GenProg to eight open source software, so that GenProg was able to fix 55 out of the 105 bugs in total [7]. After Le Goues's experiment, other researchers conducted experiments to evaluate GenProg's capability. Many other APR techniques also have been proposed after GenProg. Bug datasets created on actual bugs of open source software [8, 9] are mainly used to evaluate APR techniques including GenProg.

At this moment, there is no data on how APR techniques work on bugs in industrial software and what are issues to introduce APR techniques to industrial software development (in short, ISD). Thus, it is unclear whether applying APR techniques to ISD includes different issues applying them to open source software or not. Currently, we are conducting an academic-industrial collaboration towards introducing APR techniques to ISD. In this research, we are trying to apply GenProg and NOPOL [10] to 22 bugs that had been given from a company. We succeeded to fix a couple of bugs with the tools and found three issues inherent to ISD application. Please note that our first aim is to reveal unknown issues of applying APR techniques to ISD, not to fix bugs in industrial software as many as possible.

2 ANSWERS TO THE REQUIREMENT IN CFP

What is the new idea? Why is it new?
This is the first research that the APR techniques have been applied to ISD. This research is still on the early stage; only two tools have been applied to a single system. However, the authors believe that the current experimental results are valuable for researchers and practitioners because we found that it is not easy to introduce the APR techniques to IDS.

What is the single most related paper?
The authors consider literature [11] is the most related paper. The research applied GenProg, NOPOL, and one more tool to real bugs in open source software. As a result, GenProg and NOPOL generated one and five correct patches for 224 bugs, respectively.

What feedback do the authors expect from the forum?
The authors would like to exchange information with other researchers and practitioners who are interested in APR techniques: which tools are better? what kinds of settings work well? what kinds of new APR techniques should be effective to fix more bugs? The authors are happy if we talk

ICPC '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5714-2/18/05…$15.00
https://doi.org/10.1145/3196321.3196358

developers of any of APR techniques in person at the conference.

3 APR TECHNIQUES

In this research, currently we are using two APR tools, ASTOR [8] and NOPOL [10]. There are some reasons why we are using these tools: they can handle Java version 5, 7, and 8; they are written in Java because we need to run them in Windows environment; they are publicly available and have been used in other research [11]. Both of the tools use the Ochiai algorithm [12] to localize buggy code. ASTOR employs the GenProg algorithm [6] to modify localized buggy code. GenProg is a search-based technique while NOPOL is a semantic-based one.

There are many other techniques on APR. The search-based techniques (also known as generate-and-validate approach) search within a defined search space to generate a repair candidate and validate it with a given set of test cases. SPR [13], HDRepair [14], and Prophet [15] are classified into the search-based approach.

The semantic-based techniques synthesize a modified program by leveraging semantic information of the buggy program. Symbolic execution and constraint solving techniques are widely used in the semantic-based approach. Angelix [16], DirectFix [17], and SemFix [9] are classified into the semantic-based approach.

4 RESEARCH MOTIVATION

As shown in Section 3, many APR techniques have been proposed before now. However all of them are evaluated on bug datasets of open source software. Such bug datasets are specialized for evaluating APR techniques, so that they are detached from reality, especially from real bugs in ISD.

Thus, it is still unknown how well APR techniques work for bugs in ISD, which is our research target of this academic-industrial collaboration. Besides, if APR techniques do not work well for ISD, there may be some particular issues in ISD. We also aim to reveal what kinds of issues exist for industrial applications.

In other words, in this academic-industrial collaboration, we are researching to reveal the following research questions.

RQ1: How well do APR techniques work for bugs in ISD?
RQ2: What kinds of issues exist in ASD application?

5 RESEARCH PROCEDURE

we are conducting an experiment and an investigation.

Experiment: we are appling APR techniques to ISD.
Investigation: we have regular meetings with the company that developed the target software to discuss issues to applying the APR techniques to the target software.

Regarding the investigation, we have 1-hour meetings once/twice a month. Our university is far from the company, and so we are usually having video conferences. In video conferences, firstly we explain our progress to the company and our issues on APR application. Then, the company gives us some advice to overcome the issues. The company's advice is project-specific ones rather than APR technical ones. The authors are experts in APR techniques but not in ISD including the target software. The company has significant knowledge about the target software but not about APR techniques. This meeting is inevitable for our collaboration.

In the remainder of this section, we describe our experiment.

5.1 Experiment

The targets of our experiments are 327 real bugs found in ISD. The software was developed in Java. The software includes approximately 190 kilo lines of code.

In this experiment, we used two tools, ASTOR [8] and NOPOL [10]. Before applying the tools, we extracted a set of bugs that can be removed by modifying only source files. The followings are the steps of our bug extraction procedure. The prefix numbers mean the number of extracted bugs.

327: the bugs that we obtained from the company.
132: the bugs that were found in the process of unit testing.
78: the bugs that we could identify their fixing commits.
22: the bugs in whose modification commits only source files were modified.

Firstly, we explain why we used only bugs that had been found in the process of unit testing. In the target software development, test cases for unit testing are written with *JUnit* and they can be executed automatically. On the other hand, integration testing and system testing were conducted with scenarios. In other words, to conduct a test case of integration/system testing, a developer needs to intervene. It is difficult to run test cases of integration/system testing of the target software. ASTOR and NOPOL, which we use in this experiment, executes all test cases to generate a modified program. Thus, automated execution of test cases is mandatory in our experiment. That is why we used only bugs of unit testing. If test cases of integration/system testing were able to run automatically, we could have used the bugs in integration/system testing in our experiment. In the target software, found bugs were managed with a spreadsheet. We checked the spreadsheet and extracted the bugs that had been found in the unit testing process.

Secondly, we explain why we used only bugs that we had been able to identify their modification commits. GenProg's output is a program that passes all the test cases. However, passing all the test cases does not equal to being a correct program. If given test cases are not sufficient, a modified program passing all the test cases may still include bugs. Thus, we need to examine whether an output program is truly correct or still buggy. To conduct such examinations, developer's modification is necessary for the authors. The company gave the authors a copy of SVN repository of the target software. The authors tried to identify modification commits for each of the unit testing bugs by using bug IDs written in the spreadsheet. For some bugs in the spreadsheet, we were not able to identify their modification commit because their bug IDs did not appear in any commit logs. There are commits where multiple bugs were modified all together, so that we were not able to obtain modified code for each of the bugs. Consequently, in this filtering step, we extracted the bugs that we had been able to obtain their developers' modifications.

Finally, we explain why we used only bugs in whose modification commits only source files had been modified. APR techniques including GenProg and NOPOL can modify only source files. However, the target systems include other kinds of files such as .property files and .xml files. If such files were changed for modifying a

```
  ...
  for(int i; i < num; i++){
    if(checkNum(num)) {
      list = new ArrayList<String>();
+     break;        // Developer's modification
+     return list;  // jGenProg's modification
    }
    list.add(num);
  }
  return list;
  ...
```

Figure 1: A modification example

bug, APR techniques cannot theoretically modify it. Thus, in this experiment, we extracted the bugs in whose modification commits only source files had been modified.

Unfortunately, for most of the 22 bugs, there were no test cases to expose the bugs[1]. Consequently, we created new test cases for each of such bugs because APR techniques require a set of test cases including at least one failing one as input.

6 RESULTS

Herein, we show the results of our experiment and investigation.

6.1 Experimental results

Currently, we have succeeded to prepare a set of test cases including failing ones for 9 out of the 22 bugs. We have applied ASTOR and NOPOL to the 9 bugs. Table 1 shows our current succeeded/failed status for the target bugs. The followings are brief explanations for each status of the table.

Succeeded: the tool was able to generate a modified program.

Failed (Timeout): the tool execution was not finished within a given time-out period.

Failed (NoModificationPoint): the *Ochiai* algorithm calculated a suspiciousness value for each of the target software, but the tool was not able to decide where to modify. This error is due to tool's/environmental issues. However, more investigation is required for this error.

Failed (ExecutionException): the tool execution stopped with ExecutionException[2]. This error is due to tool's/environmental issues. More investigation is required for this error.

Failed (BugLocalization): the bug localization did not work due to insufficient test cases.

Failed (NoSolution): there was no solution in the search space of the tool.

ASTOR was able to fix a bug (BUG-1 in Table 1) in a similar way to developer's modification. Figure 1 shows ASTOR's modification and developer's modification for BUG-1[3]. The developer inserted a break-statement while ASTOR inserted a return-statement. The kinds of inserted statements are different from each other, but we can see that the two modifications are semantically the same. Moreover, the understandability of the code modified by ASTOR is reasonable. Actually, the developer of the software told us that the ASTOR's modification is acceptable. This fact means that APR techniques hold the promise of being useful for some specific bugs in ISD. However, ASTOR was not able to fix the other bugs. We shows the reasons in Subsection 6.2.

6.2 Investigation results

At this moment, we have found the following issues of APR application to ISD:

- the number of modifiable bugs,
- parameter adjustments, and
- the number of test cases.

The reminder of this subsection explains each issue in detail.

The number of modifiable bugs

As described in Subsection 5.1, there are some restrictions to apply APR techniques. The authors received 327 bugs from the company, but only 22 out of them became the targets of APR techniques. In the case of our experiment, APR techniques are applicable to only 7% bugs. The second filtering, which is extracting the bugs whose modification commits we were able to identify, is an experimental restriction, not a real one. By the second filtering, 54 bugs were removed. However, even if APR techniques can be applied to all the 54 bugs, the ratio becomes only 23%. This low ratio is the first issue of ISD application.

The fact that only 22 out of the 327 bugs are in source code is an important finding. This fact means that, we need to enhance APR techniques for other kinds of files than source files to fix more bugs. There is a technique to handle bugs in configuration bugs [18], which is a kind of interactive bug modification rather than automated program repair.

The authors carefully checked developers' modified code for the 22 bugs. As a result, we found that ASTOR and NOPOL can fix five and one out of the 22 bugs in theory, respectively. Previous studies reported that NOPOL fixed more bugs than ASTOR, but our investigation come out the opposite of the previous studies. This is because only a small percentage of bugs are due to wrong condition of if-statements.

Parameter adjustments

ASTOR and NOPOL have 67 and 15 parameter, respectively[4]. The followings are some parameters in ASTOR:

Table 1: Current succeeded/failed status for the target bugs

Bug ID	ASTOR	NOPOL
BUG-1	Succeeded	Failed (Timeout)
BUG-2	Failed (NoModificationPoint)	Failed (ExecutionException)
BUG-3	Failed (BugLocalization)	Failed (BugLocalization)
BUG-4	Failed (BugLocalization)	Failed (BugLocalization)
BUG-5	Failed (BugLocalization)	Failed (BugLocalization)
BUG-6	Failed (BugLocalization)	Failed (BugLocalization)
BUG-7	Failed (BugLocalization)	Failed (BugLocalization)
BUG-8	Failed (NoSolution)	Failed (Timeout)
BUG-9	Failed (NoSolution)	Failed (Timeout)

[1]The target software included many test cases for unit testing. But, all of them passed for the buggy versions of the target program

[2]java.util.concurrent.ExecutionException

[3]Please note that we cannot show raw source code of the target software because it is not open source software. In Figure 1, surrounding code not related to the bug and variable names are not real ones.

[4]We carefully checked all the options of ASTOR and NOPOL not to count the options that do not affect generating a modified program. For example, some options such as showing the help messages and executing the tools for the sample bugs are not included in those numbers.

- the number of variant program per generation in a genetic algorithm,
- the maximum generation in a genetic algorithm, and
- the time-out period.

We cannot easily decide optimal values for some parameters because bug modifications of APR techniques depend on target program size, the number of passing/failing test cases, the kinds of target bugs, and other elements. The purpose of using APR techniques is reducing the debugging cost, so that spending long hours to find optimal values for parameters is misplacing its priorities.

The number of test cases

The developers of the target software took count of code coverage such as statement coverage and condition coverage when they made test cases for unit testing. They also tried to achieve high coverage with a small number of test cases. This developers' strategy has poor compatibility with APR techniques. In some cases, the number of test cases was too small to apply APR techniques, so that bug localization did not work well. That is the main reason for "Failed (BugLocalization)" in Table 1. Bug localization is important because bug modification is performed for localized code. If buggy code is not localized well, it is difficult to modify it with APR techniques. The same issue occurs in open source software [19]. Consequently, the essential issue is that a sufficient set of test cases for developers to check program behavior is different from a sufficient set of test cases for bug localization of APR techniques.

6.3 Answers to RQs

From the results, we answer the two RQs as follows.

Our answer to RQ1: APR techniques hold the promise of being useful for some specific bugs in ISD.

Our answer to RQ2: We found the following three issues. The first issue is a low ratio of modifiable bugs with APR techniques. The second issue is a difficulty in adjusting parameter of APR techniques. The third issue is an essential difference between test cases for checking program behavior and for bug localization of APR techniques.

7 THREATS TO VALIDITY

We are aware that our experimental results depend the APR tools that we used. Especially, the parameter adjustment issue is clearly specific to Astor and NOPOL. However, we consider this results are enough valuable because other researchers and practitioners may apply the tools to industrial software.

To avoid personal mistake, four persons in the authors had discussions about each of the bugs to understand its symptoms and the developer's modification for it. Then, the four persons added failing test cases for the target bugs. The authors took a couple of hours for each of the bugs to add failing test cases. We referred to the existing test cases as much as possible when we create test cases. We also sent the added test cases to the developers of the software and they checked the test cases. However, if the developer added failing test cases by themselves, different test cases might have been added.

At the submission time, we have succeeded to apply the APR tools to only nine bugs, which is of course too small to draw a conclusion. We are now trying to perform the APR tools for the remaining 12 bugs. Each of the bugs requires special arrangements for the tools. For example, we need to pass JVM options ("-javaagent" and "-noverify") to run some attached test cases; however the tools have no function to pass them to GZolter, which is an implementation of the *Ochiai* algorihm and used in the tools.

8 CONCLUSION

In this paper, we reported our collaboration on introducing automated program repair techniques to industrial software development. Before now, we have succeeded to fix a real bug in a company project. On the other hand, we have revealed three issues for industrial application of automated program repair techniques. In the near future, we are going to try other automated program repair techniques. We are also going to derive techniques/methodologies to overcome/avoid the three issues.

REFERENCES

[1] J. Fishman. University of Cambridge Study: Failure to Adopt Reverse Debugging Costs Global Economy $41 Billion Annually (visited 5/Jun/2018). [Online]. Available: https://goo.gl/Dj7gC3
[2] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem Determination in Large, Dynamic Internet Services," in *DSN'02*, pp. 595–604.
[3] J. A. Jones and M. J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-localization Technique," in *ASE'05*, pp. 273–282.
[4] D. Saha, M. G. Nanda, P. Dhoolia, V. K. Nandivada, V. Sinha, and S. Chandra, "Fault Localization for Data-centric Programs," in *ESEC/FSE'11*, pp. 157–167.
[5] D. Zuddas, W. Jin, F. Pastore, L. Mariani, and A. Orso, "MIMIC: locating and understanding bugs by analyzing mimicked executions," in *ASE'14*, 2014, pp. 815–826.
[6] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically Finding Patches Using Genetic Programming," in *ICSE'09*, pp. 364–374.
[7] 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 *ICSE'12*, pp. 3–13.
[8] M. Martinez and M. Monperrus, "ASTOR: A Program Repair Library for Java," in *ISSTA'16*, pp. 441–444.
[9] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. C handra, "SemFix: Program Repair via Semantic Analysis," in *ICSE'13*, pp. 772–781.
[10] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs," *IEEE TSE*, vol. 43, no. 1, pp. 34–55, 2017.
[11] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic Repair of Real Bugs in Java: A Large-scale Experiment on the Defects4J Dataset," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, 2017.
[12] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund, "An Evaluation of Similarity Coefficients for Software Fault Localization," in *PRDC'06*, pp. 39–46.
[13] F. Long and M. Rinard, "Staged Program Repair with Condition Synthesis," in *ESEC/FSE'05*, pp. 166–178.
[14] X.-B. D. Le, D. Lo, and C. L. Goues, "History Driven Program Repair," in *SANER'16*, pp. 213–224.
[15] F. Long and M. Rinard, "Automatic Patch Generation by Learning Correct Code," in *POPL'16*, pp. 298–312.
[16] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis," in *ICSE'16*, pp. 691–701.
[17] ——, "DirectFix: Looking for Simple Program Repairs," in *ICSE'15*, pp. 448–458.
[18] A. Weiss, A. Guha, and Y. Brun, "Tortoise: Interactive System Configuration Repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 625–636.
[19] 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 *ESEC/FSE'15*, pp. 532–543.