Program Repairing History as Git Repository

Ryoko Izuta, Shinsuke Matsumoto, Yoshiki Higo, and Shinji Kusumoto
Osaka University, Japan
{r-izuta,shinsuke,higo,kusumoto}@ist.osaka-u.ac.jp

ABSTRACT

This paper proposes a concept of introducing Git repository to record a history of program evolution via automated program repair techniques. In contrast to the general usage of Git by actual developers, a Git repository is generated by an APR system. This paper presents that it is feasible to store the history of program repair efficiently and comprehensively by using Git. Moreover, the proposed concept allows to share the details of an APR execution and to compare various APR executions.

ACM Reference Format:
Ryoko Izuta, Shinsuke Matsumoto, Yoshiki Higo, and Shinji Kusumoto. 2020. Program Repairing History as Git Repository. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3387940.3392178

1 INTRODUCTION

Automated program repair (APR) is one of the promising and effective approaches to facilitate software debugging. APR is broadly classified into *generate-and-validate* (G&V) and *semantics-driven* [2]. In the search-based G&V approach, the original code is going to be evolved by repeatedly applying small code changes, and its evaluations based on a fitness function. Various studies have been conducted to improve the efficiency of G&V APR. This paper focuses on search-based G&V APR, and simply call it APR.

One of the keys to improve the APR technique is to record not only the final repairing results but also the history of program evolution. The APR history helps to analyze various APR behaviors in detail. Let us imagine a situation where a researcher conceives and develops a novel technique of program modification for APR. The developed technique should be applied to a small example to confirm whether its behavior is the same as the expected and whether the technique is effective. Such analysis requires detailed information, including source code, fitness, and parent-child relationship, for all generated variants. Besides, APR is a time-consuming process that takes several hours or even days. We believe that sharing APR histories between researchers allow reducing the time required for replicated APR experiments.

However, most APR tools do not have a feature to record the evolutionary history. The APR execution results are often limited to discovered final patches and a summary of the APR execution, which includes total execution time, the number of generated variants, the number of reached generations, and so on. For instance, our developing APR system, named kGenProg[1], cannot record the historical information. Although Astor [3], a G&V APR framework for Java, supports an option to dump all generated variants, the generated data is only a set of generated source code. At this moment, no APR tool provides fine-grained information to answer the following questions: *How were the variants generated? To what extent were variants fitted? How is a genealogy tree of variants created?*

The goal of this research is developing a framework to record and share a history of program evolution created through an APR execution. To achieve the goal, we propose a concept which employs Git to record the APR history. Generally, a version control system, such as Git, is used by actual developers to store a development history. In contrast to that, a Git repository is automatically generated by an APR system in our concept. Our concept enables the automatic recording of the program repairing history as a development history in an intuitive manner. Moreover, this concept allows to share the details of an APR execution and to compare various APR executions. There is also a possibility that some existing Git tools can be applied to analyze and visualize the APR history.

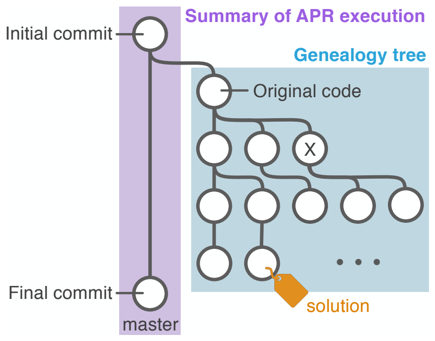
2 PROPOSED CONCEPT

There are several benefits of employing Git to record APR history. First of all, it is possible to store the entire APR process because the history of program repair is mainly made up of changes and evaluation of the source code. In addition to such historical information, initial settings (e.g., used APR tool name, tool version, and execution parameters) and final results (e.g., found patches, total execution time, and the number of generated variants) are stored in the Git repository. The initial settings allow to identify how the APR execution is conducted. APR researchers can easily conduct replication studies by sharing APR-generated information. Moreover, APR improvements can be facilitated by comparing some Git repositories which have been generated with different execution parameters.

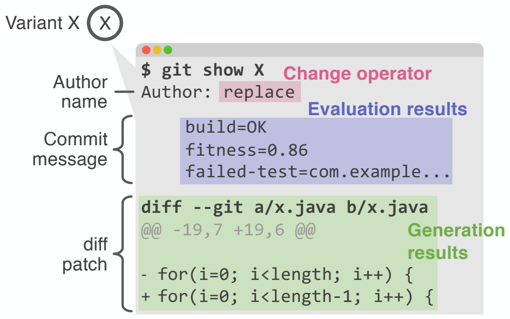
Furthermore, introducing Git will be efficient in terms of data size compared with storing plain text files. This is possible since Git only stores modified files, and APR usually applies small changes to source code files. Consequently, even though APR generates tons of variants, the difference between variants is always small.

A further benefit is that Git can be considered as *lingua franca* for APR researchers. APR system itself uses Git to manage source code [1, 3]. APR dataset, such as Defects4J, is also supplied as Git repositories. Thus, the generated repository can be manipulated intuitively for APR researchers. Another benefit of using Git is that existing Git tools such as Gitinspector can be leveraged to analyze how programs are repaired by APR techniques.

ICSEW'20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7963-2/20/05…$15.00
https://doi.org/10.1145/3387940.3392178



(a) Branch topology



```
$ git show X
Author: replace
    build=OK
    fitness=0.86
    failed-test=com.example...
diff --git a/x.java b/x.java
@@ -19,7 +19,6 @@

- for(i=0; i<length; i++) {
+ for(i=0; i<length-1; i++) {
```

(b) Details of a commit

Figure 1: Overview of introducing Git to APR

3 APPROACH

This section describes how the APR-generated information corresponds to elements in Git repository. Figure 1 shows a summary.Figure 1(a) represents a branch topology of the generated repository, and Figure 1(b) gives an example of a single commit.

The left-most branch in Figure 1(a) represents a summary of the APR execution. This branch contains two types of information: initial settings and final results. The former helps to identify and reproduce the APR execution, and the latter allows to grasp the summary of the execution. Both information is stored to a README file. This branch is labeled as a master branch to follow a common practice of Git that the master branch is used as the main development branch. Researchers can easily grasp the stored APR information by checking the README file on the master branch.

The right-side branches in Figure 1(a) represents a history of the APR execution. Each commit (each circle in the figure) corresponds to a single generated variant, and the parent of a commit corresponds to the parent of the variant. Therefore, the branch topology will be the same as a genealogical tree of the program evolution. In each variant, source code files and directory structure are stored as blob and tree objects, respectively. Thus, all changed source code is treated as code changes in general software development. Moreover, Git tags are assigned to identify variants. For example, `solution` tag is assigned to a solution variant that passes all given tests (i.e., repaired code).

Figure 1(b) shows the detail of a specific variant using `git-show` command. The change operator of each variant will be written in an author field because the operator stands for who made the variant.

```
$ cd generated-repo; git checkout master
$ cat README.md
This is an automatically generated content...
# Final results
Status: solutions found
Generated variants: 133
Total execution time: 3m19s
...
# Initial settings
randomSeed = 0
timeLimit = 60m
maxGenerations = 100
...
```

Figure 2: Checking a summary of APR execution

```
$ git diff initial solution
diff --git a/GCD.java b/GCD.java
@@ -14,2 +14,3 @@ public class GCD {
   public int gcd(int a, int b) {
-      return 0;
+     if (a == 0)
+      return b;
```

Figure 3: Checking the difference of solution

Evaluation results, such as fitness value, failed test names, and build status, are also important information. Usually, such automatically generated information is not stored in a Git repository. However, we decided to store this because they contribute to understanding APR execution. By storing the evaluation results to commit messages, users can search any variant by grep option. Furthermore, the source code difference can be shown as a diff patch.

4 USAGE

This section introduces two usages of the generated APR repository. First, the summary of the APR execution can be shown in Figure 2. We can confirm how the execution is conducted and what results are achieved by showing the README file using `cat` command. Second, Figure 3 describes a way to confirm the detailed solution of program repair. The initial and solution variants are tagged as `initial` and `solution`. Hence, we can check the difference between them by using `git-diff` command for both tags.

5 FUTURE WORKS

The implementation of this concept is left for future work. Empirical evaluation of the execution time and repository size is needed as well. Since program repair is an optimization problem, it is essential to keep one execution as short as possible.

ACKNOWLEDGMENTS

This work was supported by MEXT/JSPS KAKENHI 18H03222.

REFERENCES

[1] Y. Higo et al. 2018. kGenProg: A High-Performance, High-Extensibility and High-Portability APR System. In *Proc. Asia-Pacific Software Engineering Conference*. 697–698.

[2] L. Gazzola, D. Micucci, and L. Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67.

[3] M. Martinez and M. Monperrus. 2016. ASTOR: A Program Repair Library for Java. In *Proc. International Symposium on Software Testing and Analysis*. 441–444.