2018 25th Asia-Pacific Software Engineering Conference (APSEC)

kGenProg: A High-performance, High-extensibility and High-portability APR System

Yoshiki Higo, Shinsuke Matsumoto, Ryo Arima, Akito Tanikado,
Keigo Naitou, Junnosuke Matsumoto, Yuya Tomida, Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0871, Japan

***Abstract*—In this paper, we introduce our tool, kGenProg, which is a new automated program repair system. kGenProg has several remarkable features. Thanks to the features, kGenProg got achieved high performance, high extensibility and high portability.**

***Index Terms*—Automated program repair, Tool development, Fault localization**

I. Introduction

Automated program repair (in short, APR) has been a hot topic in the research field of software engineering during the last decade. From the viewpoint of reducing the debugging cost, APR is extreme support because exposed faults are fully-automatically fixed by APR techniques. It means that no human resources are required to fix exposed faults.

Various APR techniques have been proposed before now. A most famous technique should be *GenProg* [1]. *GenProg* leverages the genetic algorithm to find a modified version of a given faulty program. A following research reported that *GenProg* was able to generate modified versions that pass all the given test cases for 55 out of 105 faulty programs [2].

An issue of *GenProg* is that it can fix a given fault only if the fault can be fixed by reusing already-existing program statements. Another research showed that only 10% faults can be fixed with already-existing program statements in the same program [3].

There is another approach that synthesizes a modified version by leveraging semantic information of a given faulty program. Symbolic execution and constraint solving techniques are often used in this approach [4], [5]. This approach can fix faults that require non-existing program elements in faulty programs. However, due to calculation cost, this approach targets only faults in conditional predicates such as *if-statement* and *while-statement*.

Currently, the authors are developing a new system to implement and evaluate APR techniques. The developing system has the following features:

- in-memory computation,
- designed with Strategy pattern,
- high Portability, and
- visualizing the process of fault modification

In the next section, we introduce each of the above features.

II. kGenProg

We call the new sytem **kGenProg**. The system is publicly available at GitHub[1]. As its name represents, *kGenProg* is a successor APR system to *GenProg* [1] and *jGenProg* [6]. *kGenProg* (and the other two tools) firstly localize code lines that are likely causes of a given faults. Then, *kGenProg* changes the localized code lines with simple manipulations such as code insertion, deletion, and replacement. In this processing, *kGenProg* generates not *a single* mutated program but *a multitude of* mutated programs. Each mutated program is validated with given test cases. If a mutated program passes all the test cases, it is output as a modified version. *kGenProg* iterates this *generate-and-validate* process until a modified version is generated.

At this moment, *kGenProg*'s target programming language is Java because many existing tools have been developed for Java language.

A. In Memory computation

kGenProg performs the following computations in the process of APR for a given faulty program:

1) building programs,
2) inserting logging instruments to Java bytecode, and
3) executing test cases with the instrumented bytecode.

Java compiler usually outputs bytecode in the file system. However if we do so, the computation performance in 1) will not be good because a large number of compilations are performed and a bunch of file I/O happens in the iteration of *generate-and-validate*. Consequently, we designed *kGenProg* not to trigger file I/O as much as possible. In *kGenProg*, bytecode is generated as a byte array in memory. Then, the byte array gets instrumented for logging in memory. These logging instruments are required for fault localization. After that, *kGenProg* loads the classes of the byte array with a specialized class loader and performs the given test cases for the classes. In summary, *kGenProg* performs all the three computations 1), 2), and 3) in memory with a single Java VM process. Consequently, *kGenProg* get achieved a high performance to execute the iteration of *generate-and-validate* process.

[1]https://github.com/kusumotolab/kGenProg

978-1-7281-1970-0/18/$31.00 ©2018 IEEE
DOI 10.1109/APSEC.2018.00094

B. Implementation with Strategy-pattern

A big reason why the authors selected the iteration of *generate-and-validate* process of the genetic algorithm as our target is that many algorithms in the iteration can be replaced with other different algorithms. For example, currently, *kGenProg* uses *ochiai* algorithm [7] to localize faulty code. However, there are other different algorithms for fault localization. Another example is a selection of program statements for *insert* operation. *GenProg* and *jGenProg* randomly select already-existing program statements in a given faulty program. On the other hand, more recent research proposed effective strategies for selection [8], [9]. Consequently, it is important to easily switch to different algorithms and evaluate them. *kGenProg* is designed to use the Strategy pattern in the selection points so as to easily switch to different algorithms and even add new algorithms if necessary.

C. High Portability

At present, there is no APR technique that is superior to any other techniques. Thus, comparing and evaluating the differences among APR techniques. Authors of some APR tools have published their tools. However, such published APR tools are in the minority. Thus, if a research group gets a new idea of APR techniques and implements it, it is difficult to compare it with other already-proposed techniques. We have actually faced this problem. To get APR tools, we sent emails to authors of already-proposed techniques. In most cases, we received no reply or the authors said that the tools were unavailable. To promote easy-use and easy-evaluation atmosphere, we made *kGenProg*'s source code publicly available and also we made *kGenProg* as a single-jar system. Only the jar file is required to execute *kGenProg*.

D. Visualizing APR processing

We are trying to improve *kGenProg* as a practical APR tool. In such a development process, it is important to understand what kinds of program changes contribute to fix a given fault. In other words, the mutation genealogy of a given faulty program is truly informative for improving the used algorithms and inventing brand new algorithms. To obtain the genealogy information, *kGenProg* preserves information of all generated programs and their parent-child relationships. Currently, we are implementing a visualization feature for the mutation genealogy information.

III. Modification Example

Figures 1(a) and 1(b) are a simple faulty program and its test cases that are attached to the package of *kGenProg*. This program takes an integer value as input and outputs an integer value that is close to zero by 1. A fault is exposed by the second test case in Figure 1(b). If we input the faulty program and the test cases to *kGenProg*, it outputs a patch shown in Figure 1(c) within a second.

Moreover, *kGenProg* has a command line option for the number of patches to generate. If we set this option to 100, *kGenProg* generates 100 different patches for this faulty program within a minute.

```
17  public int close_to_zero(int n) {
18    if (n == 0) {
19      n++; // bug here
20    } else if (n > 0) {
21      n--;
22    } else {
23      n++;
24    }
25    return n;
26  }
```

(a) Faulty program

```
 7  @Test
 8  public void test01() { // passed
 9    assertEquals(9, new CloseToZero().close_to_zero(10));
10  }
11  @Test
12  public void test02() { // passed
13    assertEquals(99, new CloseToZero().close_to_zero(100));
14  }
15  @Test
16  public void test03() { // failed
17    assertEquals(0, new CloseToZero().close_to_zero(0));
18  }
19  @Test
20  public void test04() { // passed
21    assertEquals(-9, new CloseToZero().close_to_zero(-10));
22  }
```

(b) Test cases

```
--- example.CloseToZero
+++ example.CloseToZero
@@ -16,7 +16,6 @@
     */
    public int close_to_zero(int n) {
      if (n == 0) {
-       n++; // bug here
      } else if (n > 0) {
        n--;
      } else {
```

(c) Generated Patch

Fig. 1: A simple program modification with *kGenProg*

IV. Conclusion

In this paper, we presented a new APR tool *kGenProg*. Even *kGenProg* is still under development, we have already succeeded in modifying some real faults in the *Detect4J* dataset. In the future, we are going to invest more effective algorithms for fault localization, selection for *insert* operation and fitness function for the genetic algorithm.

References

[1] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically Finding Patches Using Genetic Programming," in *Proc. of ICSE2009*.

[2] 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 *Proc. of ICSE2012*.

[3] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The Plastic Surgery Hypothesis," in *Proc. of FSE2014*.

[4] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis," in *Proc. of ICSE2016*.

[5] 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 TSE2017*.

[6] M. Martinez and M. Monperrus, "ASTOR: A Program Repair Library for Java," in *Proc. of ISSTA2016*.

[7] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the Accuracy of Spectrum-based Fault Localization," in *Proc. of TAICPRT-MUTATION2007*.

[8] C. L. G. Xuan-Bach D. Le, David Lo, "History driven program repair," in *Proceedings of SANER2016*.

[9] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware Patch Generation for Better Automated Program Repair," in *Proc. of ICSE2018*.