Autorepairability: A New Software Quality Characteristic

Pongpop Lapvikai[*], Chaiyong Ragkhitwetsagul[*], Morakot Choetkiertikul[*], Yoshiki Higo[†]
[*]*Faculty of Information and Communication Technology, Mahidol University*, Nakhon Pathom, Thailand
[†]*Graduate School of Information Science and Technology, Osaka University*, Suita, Osaka, Japan

***Abstract*—Currently, research on automated program repair (in short, APR) is actively being conducted. APR techniques have been applied to many bugs in open-source software, but the probability of a successful fix is not very high. The authors consider that not only should APR techniques be developed, but software systems should be developed so that bugs can be easily fixed with APR techniques. In this paper, we propose *autorepairability*, a new characteristic of software quality, that shows how effective automated program repair techniques are for a specific code fragment, file, or project. We also show an approach to automatically measure *autorepairability* from the source code of a target project, and present experimental results on 1,282 Java method pairs. The use of *autorepairability* allows many studies to be conducted. For example, research on the development process for developing software systems with high *autorepairability* and research on refactoring, which transforms software with low *autorepairability* into software systems with high *autorepairability*, will be possible.**

***Index Terms*—Automated Program Repair, Program Analysis, Software Quality Model, Metrics Measurement**

I. Introduction

Automated program repair (in short, APR) is a technique to remove exposed bugs fully and automatically without human intervention. APR has become a significant research topic in the field of software engineering [1], [2]. APR techniques are often only applied to some bugs in open-source projects by the researchers who proposed them. Marginean et al. have applied their APR technique to Facebook software [3], but such examples are rare. Naito et al. claim that it is difficult to apply APR techniques to industrial projects [4].

At the moment, APR techniques can remove only a small part of bugs. Thus, if there is an index to show how effective APR techniques are in their projects, developers can decide whether they should use APR techniques in the projects or not. Such an index allows developers to avoid the situation where APR techniques hardly contribute to removing bugs that occur in their project despite the cost to them to introduce the APR techniques to the project.

ISO/IEC 25010 includes a quality model that comprehensively defines the product quality of systems and software. The product quality model comprises eight quality characteristics: Functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability. *Maintainability* is the characteristic that is related to fixing bugs, and it is defined as *degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers*. This means that *maintainability* is targeted at intended maintainers and is not specific to APR techniques. Considering the fact that there are situations where it is easier for developers to remove bugs but harder for APR techniques to remove them and vice versa, *maintainability* is not sufficient to evaluate how well APR techniques work for a software system. Consequently, another characteristic specific to APR techniques is desirable for evaluating a software system from the viewpoint of its compatibility with APR techniques.

In this paper, we introduce a new software characteristic, *autorepairability*, which means how effective APR techniques are in a specific project. We create an index to show *autorepairability*. If *autorepairability* of a target project can be easily measured, it can be used to make a decision on the introduction of APR techniques.

The following are the main contributions of this paper.

1) We present a new notion in software quality, *autorepairability*, and show that the ability to measure *autorepairability* will provide us with some new research directions.
2) We propose a simple way to measure *autorepairability* and present experimental results on a set of programs from a previous study [5].

The replication package of our study can be found at https://zenodo.org/records/10184827.

II. Motivation

As previously mentioned, *autorepairability* represents how effective APR techniques are in a specific project. In order for an automation technology to be widely used, it must be known in advance whether the automation technology is effective.

We would like to explain this concept using the following metaphor. Consider a robot vacuum cleaner (in short, RVC) as a widely used automation technology. An RVC is a machine that moves around automatically to clean. An RVC uses sensors to avoid obstacles and collects dust with brushes. One of the most significant advantages of RVC is that it automatically cleans up when you are out of the house. An RVC has a built-in battery and is cordless. When the amount of charge gets low, it returns to the charging station. Each manufacturer and model has its own unique features, such as a floor mapping function, smartphone connectivity, and resistance to step differences. However, if there is a lot of furniture and objects on the floor, an RVC cannot clean thoroughly. Depending on the manufacturer and model, there are also disadvantages such as dust being left in the corners of the room, running into furniture, not being able to return to the charging station and being stuck, and being vulnerable to step differences.

When we decide whether to install an RVC, we consider whether there are many obstacles in the room. If there are too many obstacles, we should forgo the installation of an RVC, and if there are few obstacles, we will buy an RVC.

In this way, it is crucial to accurately predict whether APR techniques will work well before it is applied. At this moment, we already know how well bugs in a project can be localized affect *autorepairability*. We fill the gap by studying what elements in a software source code affect its *autorepairability*.

The authors believe that the widespread use of *autorepairability* will lead to new research being conducted. Research on measuring *autorepairability* automatically and research on the development process to develop software with high *autorepairability* will be conducted. Moreover, if we take into account the fact that some people refrain from putting things on the floor so that RVC can work better, we can imagine that developers conduct refactorings on their projects so that APR techniques work more effectively. In this context, *autorepairability* is used as an indicator to check whether conducting refactorings are functional or not from the viewpoint of APR techniques.

III. Measuring Autorepairability

Herein, we propose a simple technique to measure *autorepairability* of a given project. The key idea is to generate a large number of artificial bugs for the given project and measure how well an APR technique can fix such artificial bugs. We use mutation testing techniques to generate artificial bugs [6], [7]. The inputs to this technique include (1) source code S_P and test cases T_P of a target project P, (2) a mutation testing technique MU, and (3) an APR tool A.

This technique includes the following three steps to measure *autorepairability*.

- **Step-1:** This technique generates mutants of source code S_P by using a mutation testing technique MU. Each mutant $\mu \in M$ is a source code that is very similar to the given source code. A large number of techniques that generate mutants have been proposed [6].
- **Step-2:** The technique applies an APR tool A to each of the generated mutants that comes with a set of test cases T_P. In this technique, each mutant μ is regarded as including a different bug as well as in mutation testing. The generated fix from A that passes all the tests in T_P is called a solution s. Set S represents all the solutions that A can generate over all the mutants in M. If a mutant passes all test cases in T_P, A is not applied to the mutant.
- **Step-3:** The technique calculates the ratio of the number of generated solutions $|S|$ per the number of mutants $|M|$, i.e., the $AR\text{-}ability(S_P, T_P, M, A)$ score. The calculation of $AR\text{-}ability(S_P, T_P, MU, A)$ is shown in Equation III.

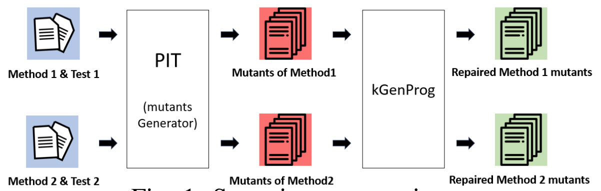


Fig. 1: Steps in our experiment

$$AR\text{-}ability(S_P, T_P, MU, A) = \frac{|S|}{|M|} \tag{1}$$

By using the $AR\text{-}ability(S_P, T_P, MU, A)$ score, we can compare the *autorepairability* of different programs.

IV. Methodology

We perform an experiment to measure the *autorepairability* of Java programs with functional equivalence. The purpose of this experiment is to show that even methods with equivalent functionality have different *autorepairability*. In other words, to show that some implementations are easy to fix bugs with APR techniques, while others are not. We also aim to reveal what kinds of program elements affect *autorepairability*.

Thus, we ask the following two research questions.

- **RQ1: What are the *autorepairability* scores of Java methods with functional equivalence?**
- **RQ2: What kinds of program elements affect *autorepairability*?**

Figure 1 shows the overall process of the experiment. We explain each part in detail below.

A. Dataset

We choose a set of 1,342 functionally equivalent Java method pairs from the Higo et al. dataset [5]. Each method pair shares the same functionality while having distinct code structures. The dataset also comes with unit test cases to ensure their functional equivalence. Thus, the two methods in each method pair pass the unit tests of each other. An example of a functionally equivalent method pair is shown in Figure 2. We can see that the `compareByteArrays` and `equals` methods similarly check two arrays whether are the same or not. However, their implementations are different.

B. Experimental Procedure

We adopt the idea of PIT [8] to generate mutants of the 1,342 functionally equivalent Java method pairs. However, mutation testing tools including PIT change bytecode directly instead of the Java source code. Thus, we developed a tool that generates mutants of the source code of a given program based on PIT. Currently, our tool includes the mutators of '*OLD_DEFAULT*' of PIT [7] since they are the most basic.

We pick the method `equals` as an example. An example mutant from the `equals` method is displayed in Figure 3 and a list of all the mutants generated from the `equals` method is

```
boolean compareByteArrays(byte[] a,int aOffset,byte[]
     b,int bOffset,int length){
  if ((a.length < aOffset + length) || (b.length < bOffset
      + length)) {
    return false;
  }
  for (int i=0; i < length; i++) {
    if (a[aOffset + i] != b[bOffset + i]) {
      return false;
    }
  }
  return true;
}

boolean equals(byte[] a,int i,byte[] b,int j,int n){
  if (a.length < i + n || b.length < j + n)   return false;
  while (--n >= 0)   if (a[i++] != b[j++])   return false;
  return true;
}
```

Fig. 2: An example method pair from the Higo et al. dataset [5]

```
boolean __target__(byte[] a,int i,byte[] b,int j,int n){
  if (a.length >= i + n || b.length < j + n)
      return false;
  while (--n >= 0) if (a[i++] != b[j++])
      return false;
  return true;
}
```

Fig. 3: A mutant of the `equals` method shown in Figure 2

shown in Table I. Looking at the mutation ID 1, we can see that the Negate Conditionals is applied to line number 5 and changes the code from `a.length < i + n` to `a.length >= i + n`. The other operations (ID 2–12) were also applied to create 11 more mutants. In this experiment, we regarded the mutated code in each mutant are buggy code and had to be fixed by an APR tool.

Then, we ran kGenProg [9] to generate solutions for each mutant. kGenProg is an APR tool of the generation-and-validation strategy as well as jGenProg [10]. The reason why we used kGenProg is its ease of use. In this experiment, we ran kGenProg with the *–max-generation* of 10 and *–headcount* of 10. The former option means how many generations kGenProg will turn the processing loop. The latter option means how many variant programs kGenProg will be produced in each generation. A hundred seed values were used to run kGenProg. In this experiment, T_P, MU, and A of $AR\text{-}ability(S_P, T_P, MU, A)$ is common to all the programs. Thus, we represent *autorepairability* of them as $AR\text{-}ability(S_P)$ for short.

Moreover, to understand more about what code structure affects the *autorepairability* of a program, the first author (who had more than 5 years of programming) performed a manual check on a sample of the method pairs. The selection criteria for selecting the sample are as follows. First, the value of the successful rate, i.e., getting a solution, after applying kGenProg of both program methods must differ by at least 50%. This is to see the differences between the two methods in a pair, which one has higher *autorepairability* and vice versa. Second, the number of mutant programs generated for each program method is more than or equal to 10. This is to have enough examples for us to observe and draw conclusions.

V. Results

A. Answering RQ1

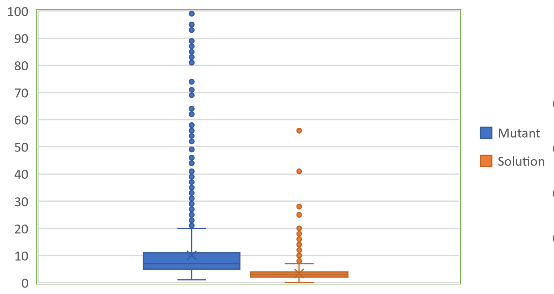
After finishing running kGenProg, we found that it could generate solutions for only 1,282 pairs. The summary of the result is shown in Table II. The lowest number of mutants is 1 while the highest is 100, with an average of 10.04 mutants per method. The lowest number of solutions that kGenProg could generate for one method is 0 while the highest is 56, with an average of 3.39 solutions per method. After we computed the $AR\text{-}ability(S_P)$ scores, we found that the average $AR\text{-}ability(S_P)$ score is 0.45, while the maximum score is 1.0 and the minimum score is 0.0. The distributions of the number of mutants and solutions are shown in Figure 4a and the distribution of the $AR\text{-}ability(S_P)$ scores is shown in Figure 4b.

TABLE II: Result of running kGenProg on 1,282 pairs

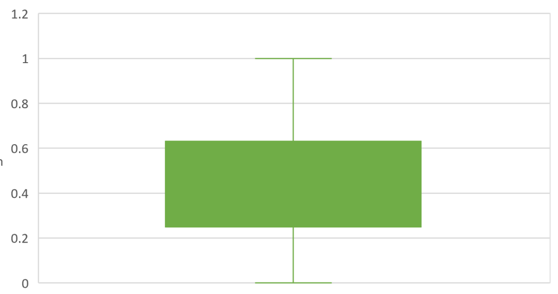
Value	Mutants	Solutions	$AR\text{-}ability(S_P)$
Min	1	0	0.0
Max	100	56	1.0
Average	10.04	3.39	0.45

TABLE I: The generated mutants of `equals` method in Figure 2

ID	Operator	Line	Original Code	Mutated Code
1	NegateConditionals	5	`a.length < i + n`	`a.length >= i + n`
2	NegateConditionals	5	`b.length < j + n`	`b.length >= j + n`
3	NegateConditionals	6	`--n >= 0`	`--n < 0`
4	NegateConditionals	6	`a[i++] != b[j++]`	`a[i++] == b[j++]`
5	Increments	6	`--n`	`++n`
6	Increments	6	`i++`	`i--`
7	Increments	6	`j++`	`j--`
8	ConditionalsBoundary	5	`a.length < i + n`	`a.length <= i + n`
9	ConditionalsBoundary	5	`b.length < j + n`	`b.length <= j + n`
10	ConditionalsBoundary	6	`--n >= 0`	`--n > 0`
11	Math	5	`i + n`	`i - n`
12	Math	5	`j + n`	`j - n`



(a) No. of mutants and solutions



(b) $AR\text{-}ability(S_P)$ scores

Fig. 4: Distribution of the no. of mutants, solutions, and $AR\text{-}ability(S_P)$ scores

TABLE III: Result of the selected pairs

Method pairs	Mutants	Success	Failed	Skipped	$AR\text{-}ability(S_P)$	Difference
`intersects(int tx, int ty, int tw, ..., int height)`	24	4	9	11	0.17	0.73
`cdRect(int sx,int sy,int sw, ...,int dh)`	10	9	0	1	0.90	
`calendarMonthToInt(int calendarMonth)`	25	2	23	0	0.08	0.69
`decodeMonth(int month)`	13	10	3	0	0.77	
`toInts(byte[] b,int off,int len)`	14	10	4	0	0.71	0.67
`toIntsVectorized(byte[] b,int off,int len)`	100	4	4	92	0.04	
`compare(Float f1,Float f2)`	10	8	2	0	0.80	0.62
`compare(Float o1,Float o2)`	11	2	9	0	0.18	
`Char(int i)`	15	11	4	0	0.73	0.60
`isXMLCharacter(int c)`	15	2	13	0	0.13	
`bilinear(double x1, double y1, ...,double z22)`	39	5	34	0	0.13	0.59
`bilinear(double x1, double y1, ...,double z22)`	39	28	8	3	0.72	
`cmod(double re,double im)`	12	3	7	2	0.25	0.57
`hypot(double a,double b)`	11	9	2	0	0.82	
`digit(int c,int radix)`	20	17	1	2	0.85	0.55
`digit(int c,int radix)`	20	6	8	6	0.30	
`quickSearch(int[] array,int value)`	11	2	6	3	0.18	0.53
`find(int[] a,int find)`	14	10	3	1	0.71	
`compare(Byte b1,Byte b2)`	10	7	3	0	0.70	0.52
`compare(Byte o1,Byte o2)`	11	2	9	0	0.18	
`hexit(char c)`	17	5	11	1	0.29	0.51
`getIntValue(char c)`	15	12	3	0	0.80	
`compare(Short s1,Short s2)`	10	6	4	0	0.60	0.51
`compare(Short o1,Short o2)`	11	1	10	0	0.09	
`compareByteArrays(byte[] a, ...,int length)`	12	10	2	0	0.83	0.50
`equals(byte[] a, ...,int n)`	12	4	5	3	0.33	

To answer RQ1, based on our preliminary study, we found that, on average, the *autorepairability* scores of Java methods in the dataset with functional equivalence [5] is 0.45. Approximately about half of the generated faults (i.e., mutants) were fixed by the kGenProg APR technique.

B. Answering RQ2

For the manual investigation, Table III shows the 11 selected pairs from the 1,282 pairs that passed our criteria and were manually checked, sorted descendingly by the differences of the $AR\text{-}ability(S_P)$ score. The method with the highest number of mutants generated is `toIntsVectorized` (100), followed by the two `bilinear` methods (39). The method with the highest $AR\text{-}ability(S_P)$ score (i.e., $\frac{|Success|}{|Mutants|}$ in this case) is `cdRect` (0.90), followed by `digit` (0.85), and `compareByteArrays` (0.83). The method pairs with the highest difference of $AR\text{-}ability(S_P)$ is `intersec` and `cdRect` with the difference of 0.73, followed by the pair of `calendarMonthToInt` and `decodeMonth` (0.69), and the pair of `toInts` and `toIntsVectorized` (0.67).

The first author performed the manual investigation by comparing the original code of the two method pairs and looking for differences in their code structures. The value in the column *difference* is the difference of the $AR\text{-}ability(S_P)$ scores. We found the following 4 code structures that affect *autorepairability* and occurred more than once in the 11 investigated pairs. We sorted them based on their number of occurrences. One method in the investigated pairs can contain multiple code structures listed below. In each example provided along with the 4 code structures below, the first code snippet always has a higher *autorepairability* score.

Code structure 1: Curly braces surrounding a code block (7 occurrences) We found that the usage of curly braces surrounding a code block can affect the *autorepairability*. For example, one method in a pair uses `if ((c>='0')&&(c<='9')) { digit=c-'0'; }` while the other method uses `if ((c>='0')&&(c<='9')) digit=c-'0';`. The former one has a higher *autorepairability* score. This means clearly specifying a code block by using curly braces can make automated program repair succeed easier.

Code structure 2: Usage of ternary operator (3 occurrences) We found that the usage of the ternary operator can offer a higher *autorepairability* score. For example, one method in a pair uses

```
return s1.shortValue()<s2.shortValue()
? -1: s1.shortValue()>s2.shortValue() ? 1: 0;
```

while the other method uses

```
if (o1.shortValue()<o2.shortValue()) return -1;
if (o1.shortValue()>o2.shortValue()) return 1;
return 0;
```

Code structure 3: Combined logical expressions in a conditional statement (2 occurrences) We found that combining logical expressions in a conditional statement can result in a higher *autorepairability* score. For example, one method in a pair uses

```
if (i >= 0x20 && i <= 0xD7FF) return true;
```

while another method uses

```
if (c < 0x20) return false; if (c <= 0xD7FF) return true;.
```

Code structure 4: Type of conditional statements (2 occurrences) We found that using `switch-case` can provide a higher *autorepairability* score than `if-else`. For example, one method in a pair uses `case Calendar.JANUARY: return 1` and another method uses `if (calendarMonth == Calendar.JANUARY) return 1`.

To answer RQ2, we found 4 code structures that affect the *autorepairability* of Java programs including curly braces surrounding code block, usage of the ternary operator, multiple conditional statements, type of conditional statements, and type of loop statement. The finding can lead to the development of automated analysis of code that will be fixed by APR whether the code contains such code structures or not to assess their readiness.

VI. Threats to Validity

The manual investigation of the code structures in 11 selected method pairs was performed by only one investigator. Thus, the result may potentially contain some human errors. We mitigated the threat by setting up a clear procedure for the comparison and identification of code structures that affect *autorepairability*. The experiment was done on the dataset of functionally equivalent Java methods [5] due to the aim of comparing and identifying code structures that affect *autorepairability*. Thus, it may not be generalized to other open-source or commercial Java projects. Lastly, we only studied kGenProg in this paper. Thus, the findings may not be generalized to other APR techniques.

VII. Implications and Conclusion

In this paper, we introduce a new characteristic of software quality, *autorepairability*. *Autorepairability* means how effective APR techniques are in a specified project. We also proposed an automated way to measure the *autorepairability* of the source code of a given program by using mutation testing techniques. We measured *autorepairability* for a total of 1,282 functional equivalent Java methods and compared their *autorepairability* scores. We manually investigated 12 pairs with high differences in *autorepairability* scores and observed code constructs that may affect the *autorepairability* scores.

Introducing *autorepairability* to the characteristics of software quality will lead to new research being conducted. For example, many methodologies of *autorepairability* measurement will be proposed. *autorepairability* at other granularity levels can be done by using high-level tests such as system tests. Methodologies of the software development process to develop high *autorepairability* software systems will be researched. Besides, *autorepairability*-based refactoring, which is a refactoring that transforms low *autorepairability* source code to high *autorepairability* one, will be researched actively.

VIII. Acknowledgement

This research was supported by the Faculty of ICT, Mahidol University, and JSPS KAKENHI Japan (JP20H04166, JP21K18302, JP21K11829, JP21H04877, JP22H03567, JP22K11985).

References

[1] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2019.

[2] "program-repair.org," http://program-repair.org/.

[3] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "SapFix: Automated end-to-end repair at scale," in *ICSE'19*, 2019, pp. 269–278.

[4] K. Naitou, A. Tanikado, S. Matsumoto, Y. Higo, S. Kusumoto, H. Kirinuki, T. Kurabayashi, and H. Tanno, "Toward introducing automated program repair techniques to industrial software development," in *ICPC'18*, 2018, pp. 332–335.

[5] Y. Higo, S. Matsumoto, S. Kusumoto, and K. Yasuda, "Constructing dataset of functionally equivalent java methods using automated test generation techniques," in *MSR '22*, ser. MSR '22, 2022, p. 682–686.

[6] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[7] "Available mutators and groups in PIT," https://pitest.org/quickstart/mutators/.

[8] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: A practical mutation testing tool for java," in *ISSTA '16*, 2016, pp. 449–452.

[9] Y. Higo, S. Matsumoto, R. Arima, A. Tanikado, K. Naitou, J. Matsumoto, Y. Tomida, and S. Kusumoto, "kGenProg: A High-Performance, High-Extensibility and High-Portability APR System," in *APSEC '18*, 2018, pp. 697–698.

[10] 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.