Constructing Dataset of Functionally Equivalent Java Methods Using Automated Test Generation Techniques

Yoshiki Higo
Osaka University
Suita, Osaka, Japan
higo@ist.osaka-u.ac.jp

Shinsuke Matsumoto
Osaka University
Suita, Osaka, Japan
shinsuke@ist.osaka-u.ac.jp

Shinji Kusumoto
Osaka University
Suita, Osaka, Japan
kusumoto@ist.osaka-u.ac.jp

Kazuya Yasuda
Hitachi, Ltd.,
Yokohama, Kanagawa, Japan
kazuya.yasuda.fd@hitachi.com

ABSTRACT

Since programming languages offer a wide variety of grammers, desired functions can be implemented in a variety of ways. We consider that there is a large amount of source code that has different implementations of the same functions, and that those can be compiled into a dataset useful for various research in software engineering. In this study, we construct a dataset of functionally equivalent Java methods from about 36 million lines of source code. The constructed dataset is available at https://zenodo.org/record/5912689.

KEYWORDS

functionally equivalent Java methods, test generation techniques

ACM Reference Format:
Yoshiki Higo, Shinsuke Matsumoto, Shinji Kusumoto, and Kazuya Yasuda. 2022. Constructing Dataset of Functionally Equivalent Java Methods Using Automated Test Generation Techniques. In *19th International Conference on Mining Software Repositories (MSR'22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3524842.3528015

1 INTRODUCTION

Since programming languages offer a wide variety of grammars, desired functions can be implemented in a variety of ways. For example, in Java, programmers can choose whether to implement iterations with `for`-statements, `while`-statements, recursive methods, or Streams. With regard to the refactoring patterns proposed by Fowler [4], both implementations before and after applying a refactoring pattern are functionally equivalent, and the refactoring can be regarded as a change in the implementation of the function. Thus, there are countless ways to implement a certain function, and programmers implement decired functions according to their own preferences and project policies.

We consider that there is a large amount of different implementations of the same functions that can be compiled into a dataset useful for various research in software engineering. For example, such a dataset can be used for evaluating code clone detection tools. Since it is desirable to detect implementations of the same functions as code clones, the performance of code clone detection tools can be evaluated by examining the extent to which different implementations with the same function are detected as code clones. We can also investigate which implementations are superior in terms of performance, such as memory usage and execution speed, and which implementations are superior in terms of software quality, such as ISO/IEC 25010:2011 [6], by using such a dataset.

In this study, we construct a dataset of functionally equivalent methods from 36 million lines of Java source code in Borge's dataset [2]. Here, functionally equivalent refers to methods that return the same output (return value) when the same inputs (arguments) are given. The key idea of this research is to automatically obtain a set of functionally equivalent method candidates by using types of return and parameters and automated test generation techniques. The authors visually checked the obtained candidate groups of functionally equivalent methods to determine whether or not they are indeed functionally equivalent. As a result, the authors identified 276 functionally equivalent method groups.

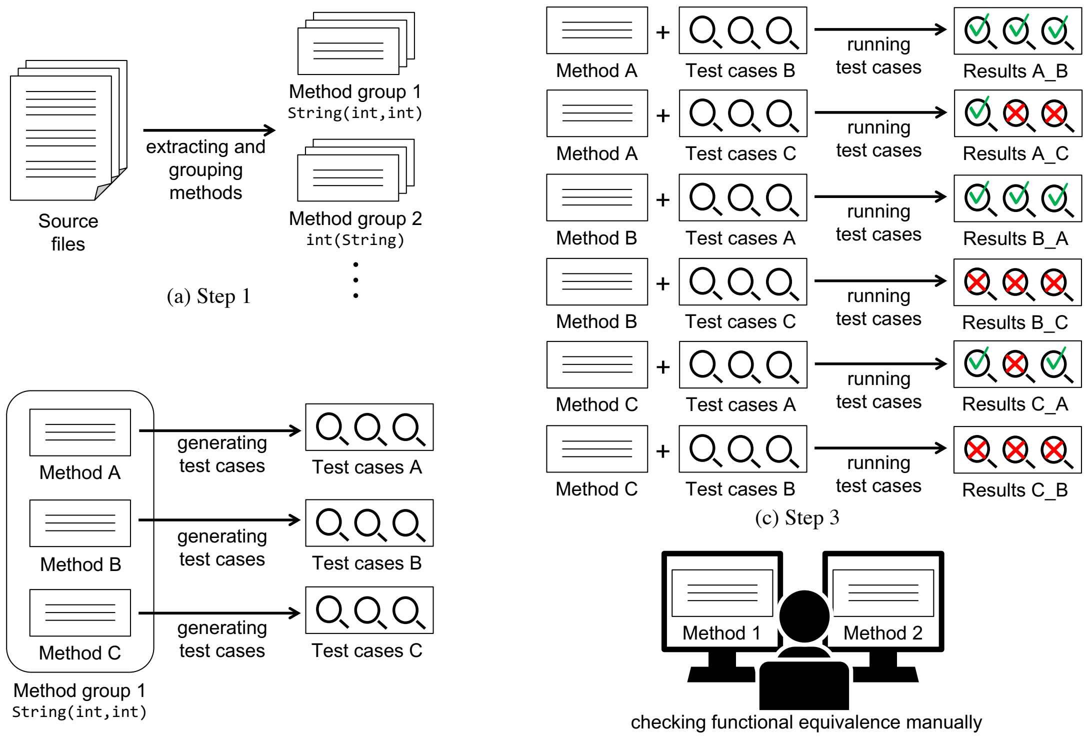
2 KEY IDEA FOR COLLECTING FUNCTIONALLY EQUIVALENT METHODS

In this research, we use the static features and dynamic behaviors of Java methods to collect candidates for functionally equivalent ones. It is necessary to visually check whether obtained candidates are really functionally equivalent. Thus, it is important to autemate the collection of candidates for functionally equivalent methods as much as possible to collect a large number of candidates.

The static features of Java methods used in this research are return and parameter types. As the first step in obtaining functionally equivalent method candidates, methods with equal return and parameter types are assigned to the same group.

The next step is to determine whether methods belonging to the same group have the same dynamic behavior by executing unit tests on them. In this study, we use an automated test generation technique to generate unit tests from a larger number of target methods. Automated test generation techniques generate tests that pass for the target method. In other words, all test cases generated from method-A pass method-A. This means that the test cases generated from method-A represent the behavior of method-A. Using

MSR '22, May 23–24, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9303-4/22/05…$15.00
https://doi.org/10.1145/3524842.3528015


Source files
extracting and grouping methods
Method group 1
String(int,int)
Method group 2
int(String)
Method A
Method B
Method C
generating test cases
Test cases A
Test cases B
Test cases C
Method group 1
String(int,int)
Method A + Test cases B running test cases Results A_B
Method A + Test cases C running test cases Results A_C
Method B + Test cases A running test cases Results B_A
Method B + Test cases C running test cases Results B_C
Method C + Test cases A running test cases Results C_A
Method C + Test cases B running test cases Results C_B
Method 1
Method 2
checking functional equivalence manually

(a) Step 1 (b) Step 2 (c) Step 3 (d) Step 4

Figure 1: Steps for collecting functionally equivalent Java methods

this property, we automatically generate tests from each method that belongs to the same group, and execute the tests mutually. If method-B passes all the tests generated from method-A and method-A passes all the tests generated from method-B, it indicates that the behaviors of method-A and method-B are equivalent to some extent, and the key idea of this research is that their functions may be equivalent if all tests run successfully against each other.

Based on this key idea, we construct a dataset of Java methods that are functionally equivalent by obtaining a set of methods that are successfully tested against each other from a large amount of open source software. Since it is obvious that methods with the same implementations are functionally equivalent, and those methods can be detected by existing code clone detection tools [5], the purpose of this research is to construct a dataset of functionally equivalent methods with different implementations.

3 PROCEDURE FOR COLLECTING FUNCTIONALLY EQUIVALENT METHODS

In this study, the following steps are used to construct a dataset of functionally equivalent methods.

STEP-1 Grouping methods included in the target projects.
STEP-2 Generating unit test cases for each method.
STEP-3 Determining the equivalence of behaviors.
STEP-4 Checking the results of STEP-3 manually.

An overview of the steps is shown in Figure 1. The first three steps are performed automatically by creating a tool. STEP-4 is conducted by the authors. Each step is described in detail below.

3.1 STEP-1

In STEP-1, the source code of the target projects is analyzed to extract methods, and the extracted methods are grouped together. In the extraction of methods, the following information is obtained for each method and registered in the database:

(1) method name,
(2) return and parameter types,
(3) original source code,
(4) normalized source code,
(5) the number of statements and conditional predicates,
(6) file path, and
(7) start and end line numbers.

In the normalization, all variables are replaced with a special name. Figure 2(b) shows an example of the normalization.

Not all methods included in the target projects are extracted. Methods satisfying any of the following conditions are ignored.

- Reference types other than `java.lang` and `lava.util` packages are used in the return type, pamareter types, and body of the method.
- The return type of the method is `void`.
- The method includes only one program statement.

The reason for using the first condition is that if a reference type other than `java.lang` package is used, it is necessary to write an `import`-statement at the top of the source file or refer to the type by its fully qualified name. Furthermore, if the type is not included in the standard Java library, it is necessary to prepare its class file (jar file), which increases the time required to compile the method.

The reason why `java.util` was chosen as an exception to the types that can be used in the methods to be extracted is that `java.util` package contains many commonly-used types such as `java.util.List` and `java.util.Set`. The number of methods that can be extracted increases dramatically by adding those types.

The reason for using the second condition is that for methods whose return value is `void`, it is difficult to determine which value is the final results of the method's calculation in an automatic manner. If the return value is not `void`, the return value of the method can be used as the final results of the method's calculation. The reason for using the third condition is that the Java source code contains a large number of setters and getters, which have very simple bodies and are not appropriate targets for methods with the same functions and different implementations.

Grouping the extracted methods is performed using the return and parameter types. Methods whose return and parameter types are exactly equal are classified into the same group. After groups are created, if there are multiple methods in the same group whose normalized source code matches exactly, only one of them is kept in the group. The reason for this is that it is obvious that methods with the same implementation have the same behavior, and such same implementations do not fit the purpose of this research. Note that a group consisting of only a single method is not subject to the processing from STEP-2.

3.2 STEP-2

In STEP-2, each method in all groups is cut into a single file to generate its test cases, and the following operations are performed when cutting the methods into files.

- Inserting '`import java.util.*;`' at the top of the file. This is a process to allow compilation even if classes of `java.util` package are used in the target method.
- Enclosing the target method in a class. At present, name '`Target`' is used. Also, changing the name of the target method to '`__target__`'. This is to ensure that all target methods are handled uniformly in the experimental script.
- Removing annotations and '`static`' from the method signatures. This is also to ensure that all target methods are handled uniformly in the experimental script.

Figure 2(c) represents the entire file when the method obtained from the source code of Figure 2(a) is extracted as a single file. From this figure, it can be seen that the extracted file contains only the target method, the class and method names are unified, and the annotations and `static` modifier attached to the method signature have been removed.

We then generate unit test cases for each of the extracted methods. Currently, we are using EvoSuite [3] to generate the test cases, but other test generation tools such as Randoop [9] and Agitar [1] can be used as well. In this experiment, any method that generated even one test case is used in STEP-3.

```
   1  package com.intellij.openapi.util.text;
   …
  33  public class StringUtil extends StringUtilRt {
   …
1265    @Contract(pure = true)
1266    public static @NotNull String repeat(@NotNull String s, int count) {
1267      if (count == 0) return "";
1268      assert count >= 0 : count;
1269      StringBuilder sb = new StringBuilder(s.length() * count);
1270      for (int i = 0; i < count; i++) {
1271        sb.append(s);
1272      }
1273      return sb.toString();
1274    }
   …
```

(a) Original source code

```
String $method(String $variable,int $variable){
  if ($variable == 0)   return "";
  assert $variable >= 0 : $variable;
  StringBuilder $variable=new StringBuilder($variable.length() * $variable);
  for (int $variable=0; $variable < $variable; $variable++) {
    $variable.append($variable);
  }
  return $variable.toString();
}
```

(b) Normalized source code

```
import java.util.*;
public class Target {
  String __target__(String s,int count){
    if (count == 0)   return "";
    assert count >= 0 : count;
    StringBuilder sb=new StringBuilder(s.length() * count);
    for (int i=0; i < count; i++) {
      sb.append(s);
    }
    return sb.toString();
  }
}
```

(c) Source code cut out to a file

Figure 2: Target Method

3.3 STEP-3

In STEP-3, we mutually execute tests on the methods that belong to the same group and have successfully generated test cases. In Figure 1(c), tests are mutually executed for Method-A, Method-B, and Method-C. Method-A passes all the tests generated from Method-B, and Method-B also passes all the tests generated from Method-A. Therefore, Method-A and Method-B are candidates for functionally equivalent methods. Method-C does not pass one of the tests generated from Method-A, and does not pass all the tests generated from Method-B. Therefore, a pair of Method-A and Method-C and another pair of Method-B and Method-C are not candidates for functionally equivalent methods.

3.4 STEP-4

STEP-4 is seeintg the source code of the candidate methods with the same behavior and different implementations obtained in STEP-3 to check whether they really have the same function.

4 CONSTRUCTED DATASET

Herein, we describe the dataset we constructed. We have constructed a dataset of functionally equivalent methods on Borge et al.'s dataset [2]. The dataset includes 2,500 projects whose source code is available on GitHub and 197 of those projects contain Java

```
int countOccurrences(String string,String substring){
  int i=0;
  int count=0;
  while ((i=string.indexOf(substring,i)) >= 0) {
    ++count;
    i=i + string.length();
  }
  return count;
}                                   https://github.com/bazelbuild/bazel
```

```
int getOccurrenceCount(String text,String s){
  int res=0;
  int i=0;
  while (i < text.length()) {
    i=text.indexOf(s,i);
    if (i >= 0) {
      res++;
      i++;
    } else {
      break;
    }
  }
  return res;
}                    https://github.com/JetBrains/intellij-community
```

```
int countMatches(String haystack,String needle){
  int num=0;
  int pos=0;
  while (pos < haystack.length()) {
    int nextPos=haystack.indexOf(needle,pos);
    if (nextPos < 0) {
      break;
    }
    num++;
    pos=nextPos + needle.length();
  }
  return num;
}                                   https://github.com/facebook/buck
```

```
int occurrencesOf(String text,String lookFor){
  int index=-1;
  int count=-1;
  do {
    count++;
    index=text.indexOf(lookFor,index + 1);
  }
  while (index != -1);
  return count;
}                                     https://github.com/neo4j/neo4j
```

Figure 3: Real examples: counting the number of times the second parameter string appears in the first parameter string

source code. Those 197 projects consist of a total of 36,316,510 lines of source code and contain 2,299,436 methods. In STEP-1, 16,936 methods were extracted and they were divided into 1,222 groups. In STEP-2, test cases were successfully generated from 5,720 methods. STEP-3 resulted in 418 candidate groups with equivalent behaviors, consisting of 1,190 methods in total.

The visual check in STEP-4 was conducted by the authors. The visual check took about eight hours, and 276 functionally equivalent groups consisting of 728 methods were obtained. The information on the obtained groups is available at Zenodo[1]. In the visual check, 109 groups were determined not to be behaviorally equivalent. In addition, 30 groups were excluded because their behaviors were equal but their implementations were not different enough to meet the condition of "equal behavior but different implementations". For example, groups that differed only in whether the variables had a `final` modifier or not, or only in the error message string, were excluded.

Figure 3 is an example of functionally equivalent methods that we found. All the four methods in the figure have the function that counts the number of times the string given as the second parameter appears in the string given as the first parameter. Method `countOccurrences` is shorter than the others but the conditional expression in the `while`-statement is complex. On the other hand, method `getOccurrenceCount` is longer, but the instructions in each line are simple and easy to understand. In addition, only method `occurrencesOf` uses `do-while`-statement.

For detailed instructions on how to use this dataset, please refer to the attached ReadMe.md file. This dataset assigns a unique ID to each method group, making it easy to obtain the source code for each method group.

[1]https://zenodo.org/record/5912689#.YfOZFfWmNqs

5 RELATED WORK

Svajlenko et al. have constructed a dataset called BigCloneBench [3, 10]. BigCloneBench is a collection of duplicate methods and the duplicate methods are classified into 45 functions. The quality of the dataset is considered to be high because they conducted visual checks at the end of the dataset construction process. However, the target of the visual check was Java methods found by a clone detection tool. Thus, the functionally equivalent methods that were not detected as code clones are not included in their dataset. In addition, the target methods were not executed during the construction process of the dataset, and functional equivalence was not determined in terms of dynamic behavior.

Liu et al. have constructed a dataset of functionally equivalent programs using past data of competitive programming [7]. They have collected functionally equivalent programs for about 5,000 problems, and the answers of several users to a certain problem in competition programming are programs with the same function. Zhao et al. publish a dataset of the same functional programs in Google Code Jam [11], and Mou et al. also publish a dataset of programs submitted to the pedagogical programming open judge system [8]. On the other hand, our dataset differs from their dataset in that it is not a set of programs in competitive programming but functionally equivalent methods included in OSS.

6 CONCLUSION

In this study, we identified functionally equivalent method groups by generating tests for methods extracted from open source software and executing the generated tests against each other. We have applied to the 197 Java projects in Borge et al.'s dataset [2]. As a result, 418 candidates of functionally equivalent method groups consisting of 1,190 methods were found. The authors visually checked all of them and identified 276 functionally equivalent method groups. The constructed dataset can be used to compare the performance of code clone detection tools and to study the quality of the code, such as understandability.

The current issue is that the execution time for finding candidates of functionally equivalent methods is very long because the tests are mutually executed for all combinations of methods whose return and parameter types are equal and for which tests could be generated. In this experiment, it took about three days only for STEP-3. In the future, we plan to introduce some heuristics to reduce the number of combinations of methods to be executed mutually, so that we can try to detect functionally equivalent method groups in a larger set of open source projects.

Another issue is that a large percentage (34%) of the method groups were determined to be functionally inequivalent by the visual investigation. In this experiment, all methods for which even a single test case was generated were targeted for dynamic behavior verification, but by limiting the number of methods for which multiple test cases were generated, the percentage of such method groups could be reduced. Also, in addition to EvoSuite, other test generation tools may be used to reduce the percentage of such method groups.

REFERENCES

[1] [n.d.]. AgitarOne. http://www.agitar.com/solutions/products/agitarone.html.

[2] Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Understanding the Factors That Impact the Popularity of GitHub Repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. https://doi.org/10.1109/icsme.2016.31

[3] Evosuite. 2021. Evosuite: Automatic Test Suite Generation for Java. https://www.evosuite.org/.

[4] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., USA.

[5] Katsuro Inoue and Chanchal K. Roy. 2021. *Code Clone Analysis: Research, Tools, and Practices.* Springer, Singapore.

[6] ISO/IEC 25010. 2011. ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models.

[7] Hui Liu, Mingzhu Shen, Jiaqi Zhu, Nan Niu, Ge Li, and Lu Zhang. 2020. Deep Learning Based Program Generation from Requirements Text: Are We There Yet? *IEEE Transactions on Software Engineering* (2020). https://doi.org/10.1109/TSE.2020.3018481

[8] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence* (Phoenix, Arizona) *(AAAI'16)*. AAAI Press, 1287–1293.

[9] Randoop. 2022. Randoop: Automatic unit test generation for Java. https://randoop.github.io/randoop/.

[10] Jeffrey Svajlenko and Chanchal K. Roy. 2016. BigCloneEval: A Clone Detection Tool Evaluation Framework with BigCloneBench. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. https://doi.org/10.1109/icsme.2016.31

[11] Gang Zhao and Jeff Huang. 2018. DeepSim: Deep Learning Code Functional Similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 141–151. https://doi.org/10.1145/3236024.3236068