Towards Automated Generation of Java Methods: A Way of Automated Reuse-Based Programming

Kento Shimonaka, Yoshiki Higo, Junnosuke Matsumoto, Keigo Naito, and Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University,
1-5 Yamadaoka, Suita, Osaka 565-0871, Japan

***Abstract*—Automatic programming has been researched for a long time. A variety of methodologies have been proposed. However, they have limited applicability, or they can generate only a few lines of code. In this research, the authors are trying to generate source code of Java methods based on their specifications. In this paper, we propose a reuse-based code generation technique with method signature and test cases. First, our technique searches existing Java methods whose signature are the same as the one input by a user. Then, our technique reworks each of them by using test cases input by the user. Methods passing all the test cases are given to the user. At this moment, the authors have implemented a naive prototype and conducted experiments with four open source software. In total, our technique succeeded to generate 18 Java methods. In this paper, we also introduce some actual examples of generated Java methods and some ideas to enhance our technique.**

I. Introduction

Automatic programming is a mechanism to generate computer programs with developer's instructions that are more abstract than program source code [1]. Automatic programming has been researched for a long time and various methodologies for automatic programming have been proposed before now. For example, Solar-Lezama proposed *sketching* [2]. In this technique, developers write partial programs in which some expressions are not written. Then, *sketching* induces the lacking expressions with SAT solver and given test cases. Gvero et al. developed a code supplement tool, *AnyCode* [3]. Developers input what kinds of functions they want to use into *AnyCode* with natural language. Then, *AnyCode* analyses users' input and suggests some functions. Gulwani et al. developed a tool, *FlashFill* [4][1], which automates string manipulation. *FlashFill* induces developer's intentions from input/output examples and performs automated string manipulations for the remaining input strings.

The following are the difficulties of automatic programming, which prevent automatic programming techniques from being used in practice:

- difficult to receive developers' intention from input [5],
- difficult to extract candidate programs [5], and
- difficult to generate non-trivial programs [6].

Currently, the authors are trying to generate Java methods from their specifications. In this paper, we propose a reuse-based automatic programming technique with the following two kinds of information:

- signature information (a list of parameter types, return type, and method name), and
- input/output information (pairs of input values and expected output values).

In general, both the two kinds of information are created in software development process. Signatures are decided in the design phase and input/output information are generated in the unit testing phase. Our technique requires only information that is created in usual software development. Developers do not have to create additional information to use our technique. Our technique has a high affinity for test-driven development.

Our technique is a reuse-based code generation methodology. First, our technique searches existing Java methods that have the same signatures as developer's input one. Then, our technique reworks found methods to pass all the input test cases. It follows that our technique generates Type-3 cloned methods.

The authors have implemented a prototype based on our technique and conducted experiments on four open source software. In the experiments, the authors firstly removed a method having test cases from the target project. Then, we tried to generate the removed method from the remaining methods in the projects. As a result, we succeed to generate 31 methods. We manually checked all the generated source code and confirmed that 18 out of the 31 methods had the correct behavior. Surprisingly, there were some generated methods whose source code was simpler than their original (hand-made) source code.

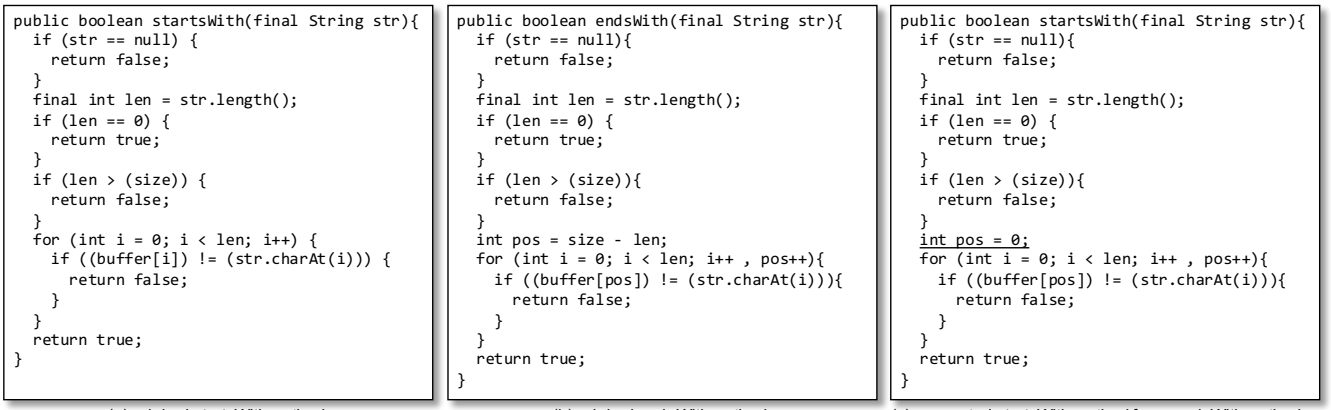
The remainder of this paper is organized as follows. In Section II, we introduce our motivating example. In Section III, we propose our technique to support the context of the motivating example. In Section IV, we introduce an automated program repair tool, *GenProg*, which being is used in our technique. Our prototype is introduced in Section V. In Section VI, the experimental results on 4 OSS are shown and then we discuss the results in Section VII. Lastly, we conclude this paper in Section IX.

II. Research Motivation

An open source project *Apache Commons Text* (in short, *commons-text*) includes two methods, *startsWith* and *endsWith*.

- *startsWith* method checks whether the contents of this object starts with the specified string or not.
- *endsWith* method checks whether the contents of this object ends with the specified string or not.

[1]https://goo.gl/PNLmfc



```
public boolean startsWith(final String str){
  if (str == null) {
    return false;
  }
  final int len = str.length();
  if (len == 0) {
    return true;
  }
  if (len > (size)) {
    return false;
  }
  for (int i = 0; i < len; i++) {
    if ((buffer[i]) != (str.charAt(i))) {
      return false;
    }
  }
  return true;
}
```

(a) original *startsWith* method

```
public boolean endsWith(final String str){
  if (str == null){
    return false;
  }
  final int len = str.length();
  if (len == 0) {
    return true;
  }
  if (len > (size)){
    return false;
  }
  int pos = size - len;
  for (int i = 0; i < len; i++ , pos++){
    if ((buffer[pos]) != (str.charAt(i))){
      return false;
    }
  }
  return true;
}
```

(b) original ends*With* method

```
public boolean startsWith(final String str){
  if (str == null){
    return false;
  }
  final int len = str.length();
  if (len == 0) {
    return true;
  }
  if (len > (size)){
    return false;
  }
  int pos = 0;
  for (int i = 0; i < len; i++ , pos++){
    if ((buffer[pos]) != (str.charAt(i))){
      return false;
    }
  }
  return true;
}
```

(c) generated *startsWith* method from *endsWith* method

Fig. 1: Motivating Example

Figures 1(a) and 1(b) show the source code of *startsWith* and *endsWith*, respectively. The two methods have not only similar functionality but also similar source code.

Herein, we assume that *startsWith* has not been implemented yet in this project. A developer wants to implement a method that checks whether the contents of the object starts with the specified string or not. If the developer knows that *endsWith* method exists in the project, she/he can do copy-and-paste and add some modifications to the pasted code for making *startsWith* method. However, if the developer does not know that *endsWith* method exists in this project, he cannot reuse *endsWith* methods. Even if he knows the existence, she/he has to search the source code in the project.

In this paper, we proposes a new technique to generate a new Java method from its specification. In the above situation where a developer wants *startsWith* method, she/he can use the proposed technique by giving the following signature information to the technique.

Method Name: startsWith
Parameter Types: String
Return Type: boolean

The developer also has to give some input/output examples. The followings are a couple of examples.

Case1: contents: abcd, string: ab, return-value: true
Case2: contents: abcd, string: abcd, return-value: true
Case3: contents: abcd, string: cd, return-value: false

Figure 1(c) is a generated *startsWith* method by the proposed technique. This method was generated based on *endsWith* method and the underlined program statement means it was changed in the generation process. This example shows that it is possible to automatically generate new methods from existing ones by adding some changes.

III. Proposed Technique

Figure 2 shows an overview of our technique to generate Java methods. As shown in this figure, our technique consists of the following two phases:

- database construction, and
- method generation.

In the database construction phase, a user specifies Java source files that are used for the method generation phase. Then, our technique analyzes the specified files to extract method information and registers the information into SQL database.

The method generation phase includes the following three steps.

Step1: searching methods
Step2: prioritizing methods
Step3: processing methods

In the method generation phase, a user input signature information and some input/output examples. Then, in Step1, our technique searches methods that have both the same parameter types and the same return type as the ones input by the user. In Step2, our technique prioritizes the found methods based on their method name similarity to the one input by the user. In Step3, our technique processes each of the found methods to satisfy all the input/output examples.

IV. GenProg

Herein, we describe *GenProg*, which is currently used in our implementation. *GenProg* is an automated program repair tool [7]. Roughly speaking, there are two main functionalities in automated program repair tools: fault localization and program modification. *GenProg* uses spectrum-based fault localization techniques [8]. In the spectrum-based techniques, every line of code is ranked based on suspeciousness value, which is calculated by executed paths of given test cases. As program modification, *GenProg* performs either of insertion, deletion, or replacement operation, randomly.

- An insertion operation means that *GenProg* randomly selects a program statement in the target project and then the selected program statement is inserted before or after the localized code fragment.
- A deletion operation means that the localized code fragment is deleted.

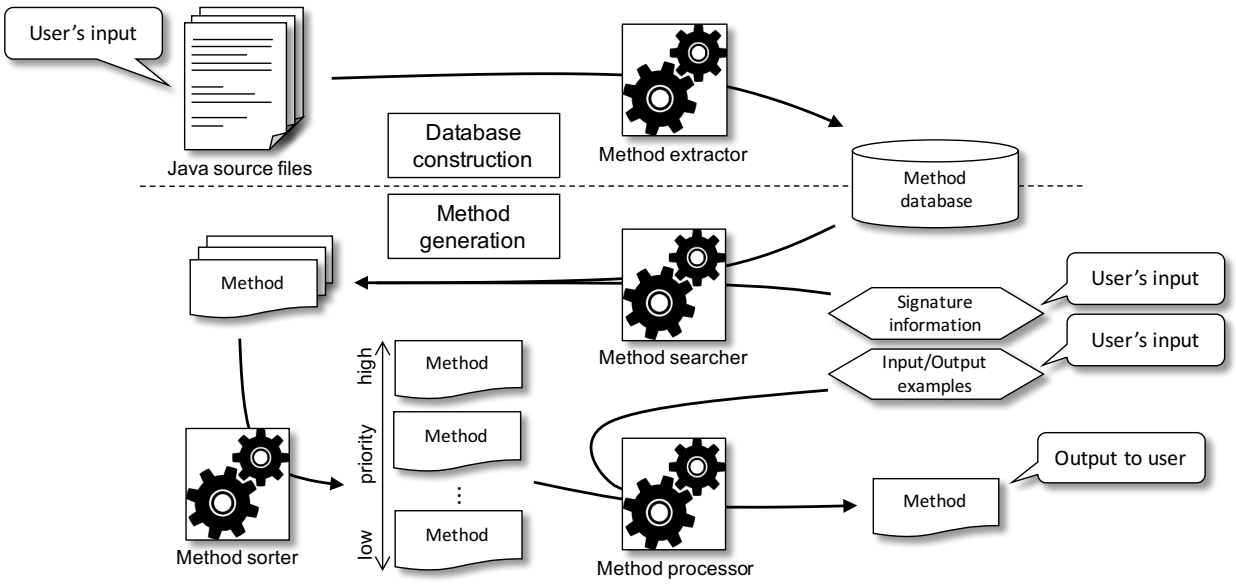


Fig. 2: Overview of Proposed Technique

- A replacement operation means *GenProg* does both the insertion and deletion operations.

The program modification process of *GenProg* is based on genetic algorithm. Several dozen of variant programs are generated by using the above operations and all the test cases are executed for all the variant programs. Then, some better variant programs are selected. Herein, "better" means passing more test cases. Then, variant programs of the next generation are generated from the selected variant programs. This process is repeated until a variant program passes all the test cases or given time limit is exceeded. Le Goues et al. applied *GenProg* to eight open source software [9] and they reported that *GenProg* had succeeded to fix 55 out of 105 actual bugs. On the other hand, *GenProg* inserts only program statements existing in the target projects, so that it cannot fix bugs that require other program statements than existing ones [10]. Another issue of *GenProg* (and other automated program repair tools) is taking a long time to fix bugs.

The first implementation of *GenProg* was realized with OCaml. But, currently, another implementation with Java is available[2]. The new *GenProg* is called *jGenProg*. *jGenProg* is included in a tool set, *ASTOR* [11]. Other automated program repair techniques, *Kali*[12] and *MutRepair*[13] are also included in *ASTOR*. In our implementaion, *jGenProg* is used.

V. Implementation

Herein, we describe our prototype implementation based on the proposed technique. The target of our prototype is Java software. Our prototype takes JUnit test cases as input/output examples. Our prototype consists of the following four components, which also are shown in Figure 2:

- method extractor,
- method searcher,
- method sorter, and
- method processor.

In the remainder of this section, we explain each of the components in detail.

A. Method Extractor

Method extractor parses given source files and extracts the following information:

- parameter types,
- return type,
- method name,
- file path,
- class name,
- project name,
- line number of method declaration, and
- source code.

Extracted information is stored into SQL database. Currently, we use *SQLite* database system[3].

B. Method Seacher

Method searcher searches the database by given signature information. All the methods whose parameter types and return type are the same as the given signature are listed. The list is passed to *Method sorter*.

C. Method Sorter

We have adopted method name similarity as the indicator of prioritizing method. Methods having more similar names are given higher priorities. This is because some studies reported that method having similar names tend to have

[2]https://goo.gl/9HzZbE

[3]https://www.sqlite.org/

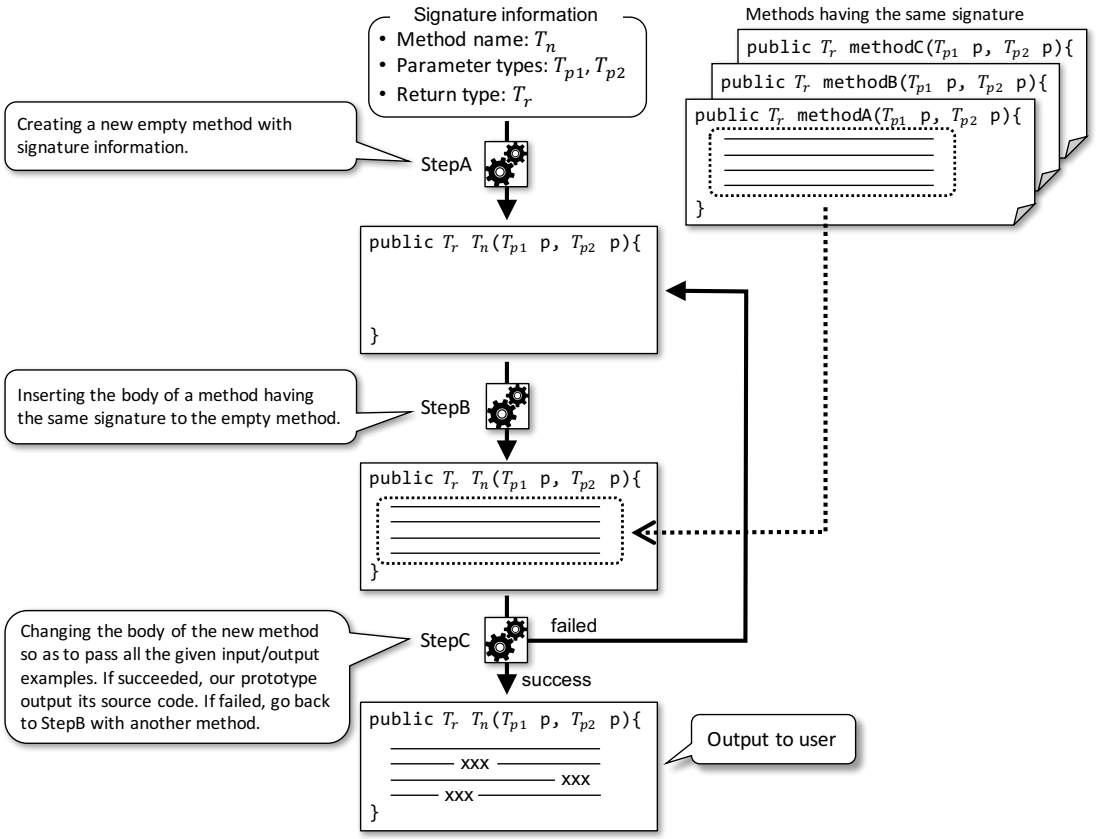


Fig. 3: Overview of method processor

similar functionalities [14], [15]. This component calculates the Levenshtein distance between input method name and each of the listed method name. The listed methods having smaller Levenshtein values have higher priority,

D. Method Processor

Figure 3 shows an overview of *Method processor*. The inputs of this component are as follows:

- signature information, and
- a method in the sorted list (in short, target method).

The procedure of *Method processor* is as follows.

StepA: creating a new empty method with the signature information.

StepB: inserting the body of a method having the same signature to the empty method.

StepC: changing the body of the new method so as to pass all the given input/output examples. If succeeded, our prototype outputs its source code. If failed, go back to StepB with another method.

Current our implementation is naive. If a compile error occurs at the end of StepB, the method is discarded. Then, *Method processor* take a new method from the list. This processing is mainly because *GenProg* requires compilable code as its input.

VI. Experiment

In this section, we report our experimental results on four open source software. The purpose of this experiment is investigating whether the proposed technique can generate plausible methods for actual software.

A. Procedure

The targets of this experiment are as follows:

- apache-commons-text,
- apache-commons-lang,
- apache-commons-io, and
- apache-commons-collections.

The reasons why we chose the above software are (1) they are being developed with Java, and (2) there are JUnit test cases in the projects.

The followings are the procedure of our experiment for each target method: we removed the source code of the selected method declaration from the software; we tried to generate the method with our technique; if we could generate, we compared the generated source code with its original one.

Before conducting the above procedure, we created a set of target methods for each software. Methods satisfying all the following conditions were included in the set.

- The method includes two or more program statements.

```
public static String removeStart(final String str, final String remove) {
  if (isEmpty(str) || isEmpty(remove)) {
    return str;
  }
  if (str.startsWith(remove)){
    return str.substring(remove.length());
  }
  return str;
}
```

(a) original *removeStart* method (developer's code)

```
public static String removeStart(final String str, final String remove) {
  if ((isEmpty(str)) || (isEmpty(remove))) {
    return str;
  }
  if (startsWithIgnoreCase(str, remove)) {
    return str.substring(remove.length());
  }
  return str;
}
```

(b) generated *removeStart* method (machine-made code)

Fig. 4: A generated method (overfitting)

```
public static boolean isBlank(final CharSequence cs) {
  int strLen;
  if (cs == null || (strLen = cs.length()) == 0) {
    return true;
  }
  for (int i = 0; i < strLen; i++) {
    if (!isWhitespace(cs.charAt(i))) {
      return false;
    }
  }
  return true;
}
```

(a) original *isBlank* method (developer's code)

```
public static boolean isBlank(final CharSequence cs) {
  if (isEmpty(cs))
    return true;
  final int sz = cs.length();
  for (int i = 0; i < sz; i++) {
    if (!isWhitespace(cs.charAt(i)))
      return false;
    }
  return true;
}
```

(b) generated *isBlank* method (machine-made code)

Fig. 5: A generated method (plausible)

- There are JUnit test cases for the method and both the statement coverage and the condition decision coverage are 100%.
- There is at least another method having the same signature as the method in the software.

The reason for the first condition is to avoid methods that are too simple to generate with our technique. Getters and setters are filtered out with this condition.

The reason for the second condition is that our technique requires test cases to generate target methods. It is unrealistic for the authors to create appropriate test cases for the methods in the target projects because the authors are not the developers of the projects. Thus, in this experiment, we targeted method having enough test cases. We used *EclEmma* to calculate statement coverage and condition decision coverage for each method in the target projects.

The reason for the third condition is that our technique can work only when there are methods having the same signature as the method that a user wants to generate.

In this experiment, we set 30 minutes as the time-out period of *GenProg*.

B. Results

Table I shows the results. "*# of generated methods*" means the number of generated methods by the proposed technique. The proposed technique generates at most a method for a given target method. All *generated methods* passes all the test cases attached to the projects. "*# of plausible methods*" means the number of plausible methods. The authors manually checked whether each of the *generated methods* was plausible or not. The authors confirmed that three out of the four projects include methods that can be generated by the proposed technique.

TABLE I: Experimental Results

Project name	Generated methods	Plausible methods
commons-text	7	5
commons-lang	18	10
commons-io	0	0
commons-collections	6	3

Figure 4 shows an *overfitting method* generated by the proposed technique. Herein, *overfitting* means a generated method passes all given test cases but it behavior does not match with developer's intention. *removeStartMethod* generated by the proposed technique (4(b)) is very similar to its original method (4(a)). However, due to the underlined instruction in 4(b), its behavior does not completely match with developer's intention. In the original code, *startsWith* method is invoked instead of *startsWithIgnoreCase*. However, the fact remains that the proposed technique was able to generate Figure 4(b)'s source code with input/output examples. The authors consider that this generated source code is helpful for developers to implement *removeStart* method. If a developer adds more input/output examples to consider case-sensitive, the proposed technique can generate a method that is identical to the original method.

Figure 5 shows a case where the proposed technique was able to generate a *plausible* method. We can see that the machine-made code is shorter/simpler than its developer's code. The underlined instruction in the machine-made code is different from the developer's code. In the developer's code, null-checking and zero-length checking are performed directly. The developer might not know of *isEmpty* method. On the other hand, the proposed technique generated code by using *isEmpty* method. The content of *isEmpty* method is null-checking and zero-length checking, so that the generated method is semantically the same as its developer's code. This case shows that the proposed technique has a capability of generating better code than developers.

VII. Discussion

We investigated which ranks of methods had been bases of generated methods. In other words, we investigated whether our method sorting strategy (described in Subsection V-C) worked well or not. Figure 6 shows that the ratio of A

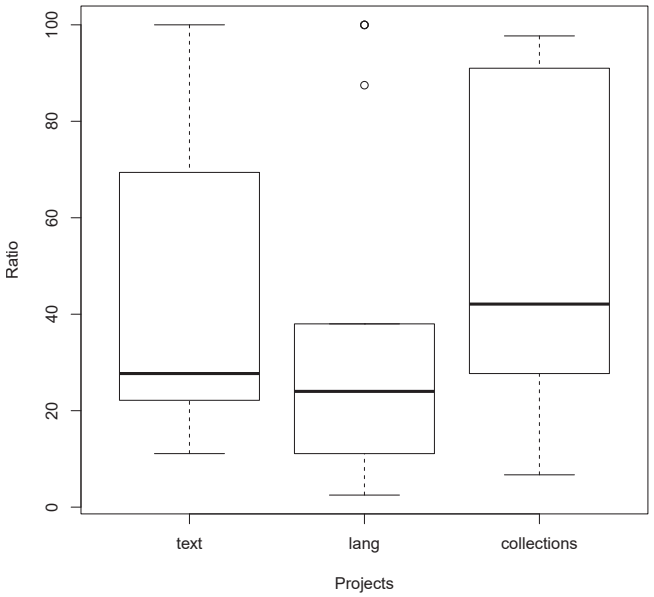


Fig. 6: Ratio of processed methods against methods having the same parameter types and return type as user's input ones

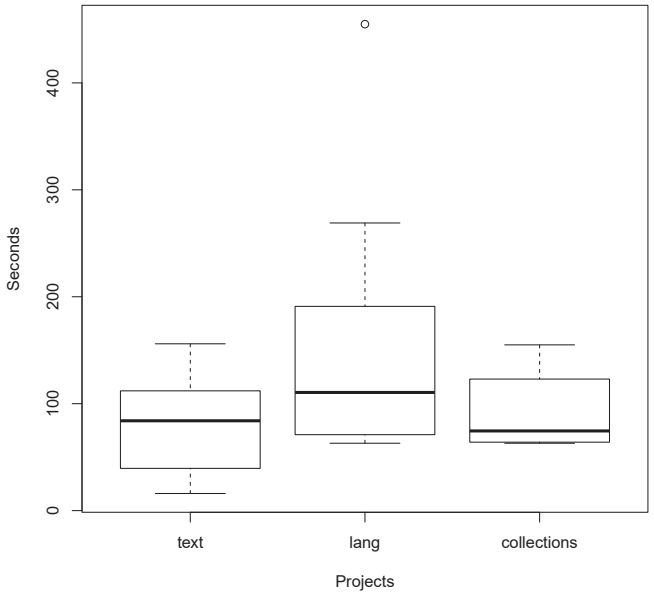


Fig. 7: Execution Time

against B: A means the number of processed methods before generating a method passing all the test cases; B means the number of methods having the same parameter types and return type as user's input ones. For all the three projects where the proposed technique succeeded to generated some methods, median values are 27%, 24%, and 42%. This means that our sorting strategy works well. Using method name similarity to prioritize methods to process is useful to shorten generation time.

Next, we discuss generation time. There are theoretically two reasons why the proposed technique fails to generated methods. The first reason is that the execution time of our prototype reached 30 minutes, which is time-out period. The second reason is that there is no answer in the search space of our technique.

Figure 7 shows execution time of the prototype where the prototype was able to generate *overfitting* or *plausible* methods. For all the cases, methods were generated within much shorter time than 30 minutes. These results mean that using longer time-out period is not useful. If we use other algorithms than *GenProg*, we may succeed to generate more methods.

As described in Subsection V-D, changing the body of a given method can start only if the given method is compilable. This is because GenProg requires compilable code as its input. The authors consider it is a major drawback to method changing because uncompilable methods are unavailable even if they almost satisfy the given input/output examples. Currently, we are trying to change implementation to start with uncompilable methods.

The experiment was conducted under the assumption that the existence of good test suite. However, it may not be realistic. At the beginning, the developer may have only vague understanding of the method. Furthermore, it is not relatively easy to generate test cases for methods taking only simple data types (e.g., int or String)., but it can be hard for object-type inputs. Thus, the experiment threats to construction validity. Other kinds of experiments such as using partial test cases or user interaction are necessary for fair evaluations.

At the end of the discussion, the authors have to say that the proposed technique generates Type-3 clones in the target software. If a buggy method is reused by our technique, a new method is generated based on the buggy code. However, if plenty of input/output examples are given, the bugs in the code are automatically removed in the phase of method processing of the proposed technique.

VIII. Related Work

Reiss has proposed a reuse-based Java method generation techniques [16]. The technique takes method's specification as its input and output a method that satisfying the specification. Firstly, the technique searches existing methods that include given keywords or match with given signature restrictions. Then, the technique transform the found methods to pass all the given test cases. The transformations can be divided into some groups.

- The first group is signature transformation. The return type, parameter types, and method name of a found method are changed to meet the given signature. Reordering parameters, adding/changing/deleting exceptions, and converting to *static* methods are also performed.
- The second group is generative transformation. This transformation is to find a code fragment that satisfies the given specification. This transformation (extraction) is realized with backward slicing techniques.
- The third transformation is compilation transformation. Program code where compilation errors occur is removed.
- The fourth group is testing transform. This transformation looks at the test results and checks if there is a simple

transformation that will convert the actual results into the desired ones. Inverting boolean returns, changing the case of string results, and adding a constant value to integer results are performed.

The biggest difference between our technique and Reiss's one is the way to transform methods. Reiss's method changes signatures, extracts a code segment, removes uncompilable program statements, and add some small changes to pass test cases. On the other hand, our technique reuses program statements in the same or other methods in the same projects. Our technique make larger methods by the transformation than their base methods while Reiss's technique make smaller methods than their base methods.

Stolee proposed to use a SMT solver to find a match against existing programs instead of executing test cases [17]. A remarkable feature of this technique is a capability of composing multiple methods if no single database meet a given specification. However, this technique does not change code inside existing methods to meet the given specification.

Lazzarini Lemos et al. developed an Eclipse plugin, *CodeGenie* [18]. This tool searches existing code based on test cases, which are given by developers. This tool performs program slicing to eliminate code fragments that are not related to given test cases. However, this technique do not have the capability to change existing code to satisfy the given test cases.

IX. Conclusion

In this paper, we proposed a technique to generate Java methods from given parameter types, return type, and input/output examples. We have implemented a prototype based on the prototype technique and applied to four open source software. As a result, the prototype generated 31 methods and 18 out of them are plausible.

We have many future works. First of all, we need to conduct a deeper analysis of the experimental results. We need to reveal why the proposed technique did not generate any method for apache-commons-lang. We are going to do more experiment with this prototype. For example, we are going to investigate whether our technique can generate Java methods that were added in revision *r+1* with the project source code of revision *r*. Experiments with partial test cases or user interaction are also necessary.

We understand that the applicability of our technique is limited because it requires the same signature methods for generating methods. We are going to extend our technique to transform signatures like Reiss's technique [16].

We are also going to use other algorithms to generate Java methods. Currently, our prototype includes *GenProg*, but in the near future, we add *NOPOL* [19], which is another automated program repair algorithm with SMT solver. Moreover, we may produce new algorithms that fit more closely to our research context.

Acknowledgment

This work was supported by MEXT/JSPS KAKENHI 25220003 and 17H01725.

References

[1] R. A. Mur, "Automatic inductive programming," in *The 23rd international conference on Machine learning, Tutorial*, 2006.

[2] A. Solar-Lezama, "Program Sketching," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5-6, pp. 475–495, 2013.

[3] T. Gvero and V. Kuncak, "Synthesizing Java Expressions from Free-form Queries," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015, pp. 416–432.

[4] S. Gulwani, W. R. Harris, and R. Singh, "Spreadsheet Data Manipulation Using Examples," *Communications of the ACM*, vol. 55, no. 8, pp. 97–105, 2012.

[5] S. Gulwani, "Dimensions in Program Synthesis," in *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, 2010, pp. 13–24.

[6] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "DeepCoder: Learning to Write Programs," in *5th International Conference on Learning Representations*, 2017.

[7] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A Generic Method for Automatic Software Repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[8] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. j. J. C. van Gemund, "A Practical Evaluation of Spectrum-based Fault Localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[9] 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 *Software Engineering (ICSE), 2012 34th International Conference on IEEE*, 2012, pp. 3–13.

[10] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The Plastic Surgery Hypothesis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 306–317.

[11] M. Martinez and M. Monperrus, "ASTOR: A Program Repair Library for Java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 441–444.

[12] Z. Qi, F. Long, S. Achour, and M. Rinard, "An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 24–36.

[13] V. Debroy and W. E. Wong, "Using Mutation to Automatically Suggest Fixes for Faulty Programs," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, 2010, pp. 65–74.

[14] A. Corazza, S. D. Martino, V. Maggio, and G. Scanniello, "Investigating the Use of Lexical Information for Software System Clustering," in *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, 2011, pp. 35–44.

[15] Y. Higo and S. Kusumoto, "How Should We Measure Functional Sameness from Program Source Code? An Exploratory Study on Java Methods," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 294–305.

[16] S. P. Reiss, "Semantics-based Code Search," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 243–253.

[17] K. T. Stolee, "Finding Suitable Programs: Semantic Search with Incomplete and Lightweight Specifications," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 1571–1574.

[18] O. A. Lazzarini Lemos, S. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes, "A Test-driven Approach to Code Search and Its Application to the Reuse of Auxiliary Functionality," *Information and Software Technology*, vol. 53, no. 4, pp. 294–306, 2011.

[19] J. Xuan, M. Martinez, F. DeMarco, M. i. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2017.