Generating Clone References with Less Human Subjectivity

Yusuke Yuki*, Yoshiki Higo*, Keisuke Hotta* and Shinji Kusumoto*
*Graduate School of Information Science and Technology, Osaka University, Japan
Email: {y-yusuke, higo, k-hotta, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—In evaluating code clone detection tools, a benchmark is created to measure their precision and recall. Benchmarks in previous research have either of the following issues: the first one is that they depend on the code clone definitions of benchmark creators; the second one is that they are not code clones occurring in actual development process. To get rid of both the two issues, we propose a methodology that creates code clone references based on code clones occurring in development process without any human judgements. More concretely, we use multiple revisions included in the source code repository of target software to identify merged methods in the past development process. We regard merged methods as real code clones. The authors' benchmark can evaluate detection accuracy of code clone detection tools more objectivity.**

***Index Terms*—code clone, clone detection tool, evaluating clone detection tools, benchmark, merging methods**

I. Introduction

A code clone (hereafter, clone) is a code fragment that is similar or identical to another code fragment in source code. Copy-and-paste operation in code implementation is the main reason of clone occurrences [1] [2]. It is said that the presence of clones makes software maintenance more difficult. For example, if a code fragment includes a bug, we need to check its clones too. For such a reason, it is important to understand how clones are distributed in source code. A variety of clone detection tools have been developed before now [1] [2]. Thus, we need to evaluate which clone detection tools have better detection capabilities than others.

In evaluating clone detection tools, a benchmark (a set of real clones) is created to measure their precision and recall [3] [4] [5]. Precision means a ratio of real clones included in detected ones. Recall means a ratio of real detected clones against all the real clones in given source code. Consequently, the evaluation results depend on benchmarks. If unreal clones are included in a benchmark, those unreal clones have negative impacts on measurements of recall and precision. Hereafter, we call real clones *clone references*.

There are several research studies that created clone benchmarks. All of the studies created benchmarks with either of the following procedures.

Judging with eyes: researchers judged similar code fragments with their eyes one by one [3] [6]. Benchmarks created with this procedure are probably influenced by subjectivity of benchmark creators. Clone references in such benchmarks depend on the clone definitions of benchmark creators, so that objectivity of precision and recall measurement is not sufficient. Besides, real clones that were not detected by clone detection tools are not included in such benchmarks.

Creating artificial clones: researchers extracted some code fragments randomly from target source code, then, added certain changes to code fragments, and inserted the changed code fragments to randomly selected positions of the target source code [7]. A code fragment and its changed version are supposed to be reported as clones. In other words, they are regarded as clone references. An issue of this procedure is that clone references are not clones occurring in actual development process. There were some cases that artificial clone references had negative effects on accuracy measurement [4].

To get rid of both the two issues, we propose a methodology to create clone references based on clones occurring in development process without any human judgements. More concretely, we use multiple revisions included in the source code repository of target software to identify merged methods in the past development process. Merging methods is a way to remove duplicated methods. We believe that at least such a pair of methods merged in the past should be detected as clones. In other words, merged methods should be clone references. If we detect merged methods in an automated way, we can create a large number of clone references not depending on human subjectivity.

We have developed a tool based on our methodology and applied it to some open source software. Then, we manually investigated tool's results one by one. As a result, we confirmed all of the tool's outputs (19 groups of methods) were real clones. Our clone references is available at our website[1].

The main contribution of this research is as follows.

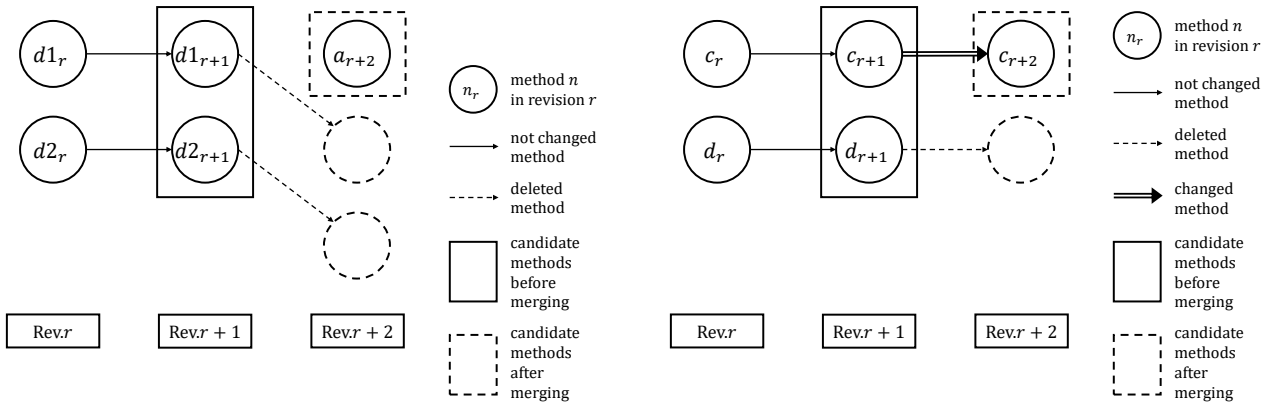
- We resolved the two issues lying in existing benchmarks. In other words, we proposed an automated way to create clone references occurring in actual development process.
- We applied the tool implemented based on the proposed way to some open source software, and we confirmed that all the created clone references were real clones.

II. Benchmarks in Previous Research

Bellon's benchmark [3] was created in the following procedure: firstly, clones were detected from Java and C software with six clone detection tools; then, 2% of detection results were manually checked whether they were real clones or not one by one. The authors think that there are two issues in Bellon's benchmark.

- Acceptance/rejection decisions depend on Bellon's subjectivity. If someone else does the same check for the same clones, decision results will be different.

[1] http://goo.gl/9Mhwo9



(a) Pattern1 (b) Pattern2

Fig. 1. Two Patterns of Merging Methods

- The benchmark depends on clone detection results. Clone references in Bellon's benchmark is limited to clones detected by detection tools in the year 2002.

Mutation and Injection Framework (in short, MIF) [4] [7] created clones artificially. MIF can create clone references without being affected by human subjectivity. However, it has the following issue.

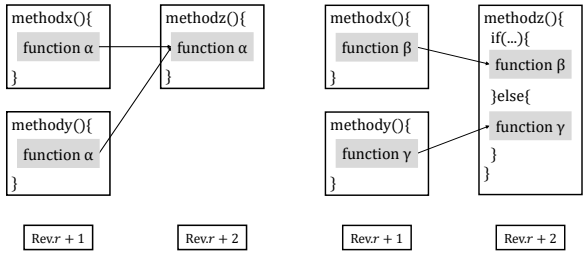
- Clone references in MIF are not clones occurring in actual development process. Clone references in MIF are occasionally clones that are not likely to occur in practice. For this reason, there are some cases where precision and recall measurements may not have been able to be done accurately.

BigCloneBench [5] [6] selected well-known functions such as bubble-sort or file-copy as its clone reference creation targets. Firstly, code fragments that might have any of well-known functions were extracted by using the features of the well-known functions. Secondly, each of the code fragments was manually checked whether it really had the function. Accepted code fragments are clone references in the benchmark. In this methodology, clone references can be created without using clone detection tools. BigCloneBench includes the following two issues.

- In acceptance/rejection decisions, human subjectivity probably affected clone reference creation results. If other researchers did the same work, difference sets of clone references would have been created.
- Candidates of clone references are limited to well-known functions appearing in source code.

III. The Authors' Benchmark

Benchmarks in previous research have either of the following issues: the first one is that they depend on the clone definitions of benchmark creators; the second one is that they are not clones occurring in actual development process. In this study, we propose a new methodology to create benchmarks which can solve both the issues.



(a) Merging Clones (b) Not Merging Clones

Fig. 2. A Example of Calculating the Similarity

A. Key idea

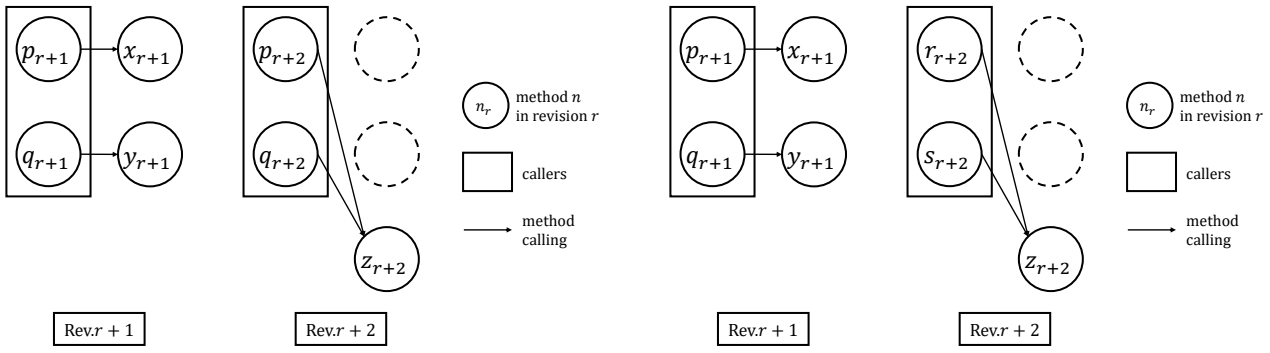
Authors in previous research use a single revision of projects to make their benchmarks. On the other hand, we use multiple revisions included in the source code repository of target software. More concretely, we detect merged methods in the past development process. Merging is one of refactoring techniques against clones. We regard merged methods as clone references. By detecting merged methods automatically, we can acquire many clone references with less human subjectivity.

B. Abstract of the authors' benchmark

We define two patterns of merging methods. After detecting merged methods candidates by using the two patterns, we calculate a similarity between candidate methods before and after merging and use information of method calling to eliminate false positives.

Pattern1: in this pattern, developers merge multiple deleted methods into an added method. An example is shown in Figure 1(a). In Figure 1(a), methods $d1$ and $d2$ were deleted between revisions $r+1$ and $r+2$. Method a was added between revisions $r+1$ and $r+2$. Hence, methods $d1$ and $d2$ are candidate methods before merging and method a is a candidate method after merging.

Pattern2: in this pattern, developers merge multiple deleted methods into an existing method. An example is shown in Figure 1(b). In Figure 1(b), method d was deleted between



(a) Merging Clones

(b) Not Merging Clones

Fig. 3. A Example of Using Method Calling

revisions $r+1$ and $r+2$. Method c was changed between revisions $r+1$ and $r+2$. Hence, methods d and c in revision $r+1$ are candidate methods before merging and method c in revision $r+2$ is a candidate method after merging. We target a changed method because a method is changed to add parameters or branch conditions when merging.

After detecting merged methods candidates by using the two patterns, we calculate a similarity between candidate methods before and after merging and use information of method calling. The reason we calculate the similarity is that methods having the same function are merged when merging duplicated methods. An example is shown in Figure 2. In Figure 2(a), methods x and y in revision $r+1$ had the same function α. Subsequently, method z in revision $r+2$ had the same function α. This case represents merging clones. In Figure 2(b), methods x and y in revision $r+1$ had different functions β or γ. Subsequently, method z had both functions β and γ in revision $r+2$. This case does not represent merging clones. For this reason, we calculate the similarity. If there are multiple candidate methods before merging which are greater than or equal to the threshold, we regard them as true positives. We calculate the similarity between different revisions, which is different from calculating the similarity in detecting clones.

The reason we use information of method calling is that methods calling deleted methods call an added method instead of the deleted methods when merging clones. An example is shown in Figure 3. In Figure 3(a), method x in revision $r+1$ was called by method p and method y in revision $r+1$ was called by method q. Subsequently, method z in revision $r+2$ was called by methods p and q. This case represents merging clones. In Figure 3(b), method x in revision $r+1$ was called by method p and method y in revision $r+1$ was called by method q. Subsequently, method z in revision $r+2$ was called by methods r and s. This case does not represent merging clones. For this reason, we use information of method calling. If multiple candidate methods before merging share at least the same method calling with a candidate method after merging, we regard them as true positives.

C. Steps of making the authors' benchmark

We explain steps of making benchmarks with the proposed methodology. The input is the source code repository of target software. The output is information of merged methods.

STEP1: we detect deleted, added and changed methods between every pair of consecutive revisions

STEP2: we relate candidate methods before merging to candidate methods after merging by using the two patterns.

STEP3: we calculate the similarity between candidate methods before and after merging

STEP4: we detect merged methods by using information of method calling.

We execute the above steps for all revisions of the repository automatically with our prototype tool. In this paper, detailed explanations are omitted due to space limitations.

IV. Characteristics of the Authors' Benchmark

A. How to use the authors' benchmark

We use not a single revision but multiple revisions to make the authors' benchmark, and so researchers evaluating clone detection tools need to execute the detection tools against all Java source files in the revisions where clone references exist.

B. Usability of the authors' benchmark

By using the authors' benchmark, researchers can evaluate detection tools better objectivity than ones created by existing methodologies because this benchmark does not depend on our subjectivity to judge whether it is clone or not and clone references are clones occurring in actual development process.

C. Limitations of the authors' benchmark

We explain limitations of the authors' benchmark in the following three points.

First, clone references in this study are limited to merged clones. Merged clones are only a small part of clones to be detected. However, clone detection tools detect other kind of clones than merged clones. Thus, for each of the detected clones which are not included in clone references, researchers cannot judge whether it is another kind of clones or a false positive. Also, clone references in this study are a part of real

TABLE I
OVERVIEW OF TARGET SOFTWARE AND RESULTS

Software	Target directory	Start revision(date)	End revision(date)	# of target revisions	# of detected methods
Ant	/ant/core/trunk/main	r267549(2000/01/13)	r1240680(2012/02/05)	6,022	7
ArgoUML	/trunk/src	r1(1998/01/27)	r19893(2012/07/10)	3,925	10
jEdit	/jEdit/trunk	r3791(2001/09/02)	r22016(2012/08/17)	5,168	2

clones in target projects, so it is difficult to measure precision with the authors' benchmark.

Second, the granularity of clone references is limited to the method-level. We suffered from finding a large number of clones. However, there is a case where a part of a method is merged. As future work, we are going to extend the authors' benchmark to handle other granularities of clones.

Third, the number of clone references in our benchmark is small. Evaluation results of detection tools with our benchmark may lack generality. However, we assigned the priority to reduce false positives. We can extend the current definition of clone references by detecting other refactorings on clones in the source code repository of target software.

V. EVALUTION

A. Setup

We have implemented a software tool based on the proposed methodology. Currently, targets of our implemented tool are limited to software systems managed by using Subversion and written in Java. We specified 50 tokens and 6 lines as thresholds of minimum clone length, which means that every clone must have at least 50 tokens and at least 6 lines. We specified 70% as the threshold of a similarity between candidate methods before and after merging. Those are one of often-used thresholds in clone detection [3] [5].

In this experiment, we selected Ant, ArgoUML, and jEdit as the targets. Table I shows an overview of target systems. We selected them because they are popular and successful systems. The number of target revisions represent the number of revisions where source code was added, deleted or changed. The number of detected methods represents the number of groups of merged methods.

B. Results

Nineteen groups of merged methods were detected, and we manually checked all of the source code one by one. As a result, we found that all the detected methods were merged methods.

C. Threats to validity

Evaluation methodology: in this experiment, we manually investigated whether detected methods were clones or not. However, investigation results may not be entirely correct because the authors are not developers of the target systems.

Target systems: in this experiment, we targeted only three systems. If we had selected other systems, the results might have been different from this experiment.

Thresholds: in this experiment, we used thresholds (50 tokens, 6 lines, and 70% similarity) which generally used in detecting clone, and so we reduced our subjectivities as much as possible. We should explore other numerical values as thresholds.

VI. CONCLUSION

In this paper, we proposed a new methodology to create clone benchmarks. The methodology leverages multiple revisions in a source code repository and detects merged methods in the past development. Detected merged methods are regarded as clone references. Benchmarks created with our methodology have the following benefits.

- The benchmarks have better objectivity than ones created by existing methodologies.
- Clone references in the benchmarks are clones occurring in actual development process.

We have implemented a software tool based on the proposed methodology and applied it to several open source software. Nineteen groups of merged methods were detected, and we manually checked all of the source code one by one. As a result, we found that all the detected methods were actual duplicated methods.

The benchmark is useful to evaluate detection accuracy of clone detection tools.

In the future, we are going to conduct more experiments to other large software systems. We are also going to extend the methodology to handle more fine-grained clones such as block-level. Finally, we are going to evaluate many of the state-of-the-art detection tools with benchmarks created by our extended methodology.

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Number 25220003.

REFERENCES

[1] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[2] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *Software Engineering, IEEE Transactions on*, vol. 33, no. 9, pp. 577–591, 2007.

[4] J. Svajlenko and C. K. Roy, "Evaluating modern clone detection tools," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 321–330.

[5] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *2015 IEEE International Conference on Software Maintenance and Evolution*, 2015, pp. 131–140.

[6] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 476–480.

[7] J. Svajlenko, C. K. Roy, and J. R. Cordy, "A mutation analysis based benchmarking framework for clone detectors," in *the 7th International Workshop on Software Clones*, 2013, pp. 8–9.