A Technique to Detect Multi-grained Code Clones

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

***Abstract*—It is said that the presence of code clones makes software maintenance more difficult. For such a reason, it is important to understand how code clones are distributed in source code. A variety of code clone detection tools has been developed before now. Recently, some researchers have detected code clones from a large set of source code to find library candidates or overlooked bugs.**

In general, the smaller the granularity of the detection target is, the longer the detection time. On the other hand, the larger the granularity of the detection target is, the fewer detectable code clones are.

In this paper, we propose a technique that detects in order from coarse code clones to fine-grained ones. In the coarse-to-fine-grained-detections, code fragments detected as code clones at a certain granularity are excluded from detection targets of more fine-grained detections. Our proposed technique can detect code clones faster than fine-grained detection techniques. Besides, it can detect more code clones than coarse detection techniques.

***Index Terms*—multi-grained detection technique, file-level clone, method-level clone, code-fragment-level clone**

I. Introduction & Related Work

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]. 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 has been developed before now.

Recently, some researchers have detected clones from a large set of source code to find library candidates or overlooked bugs [3]. Merging the same function in multiple software into a library is beneficial from the viewpoint of development efficiency improvement.

The existing clone detection techniques detect only single granularity clones, such as file-level ones or code-fragment-level ones. The existing clone detection techniques can be roughly classified into the following two detection techniques [4].

Coarse detection technique: detecting files, methods, and blocks as clones.

Fine-grained detection technique: detecting any code fragments as clones. Some clones are large but others are not so large. Any size duplicated code fragments are found as clones if they are larger than a given threshold. Fine-grained detection techniques can detect clones whose code fragments are parts of classes, methods, or blocks.

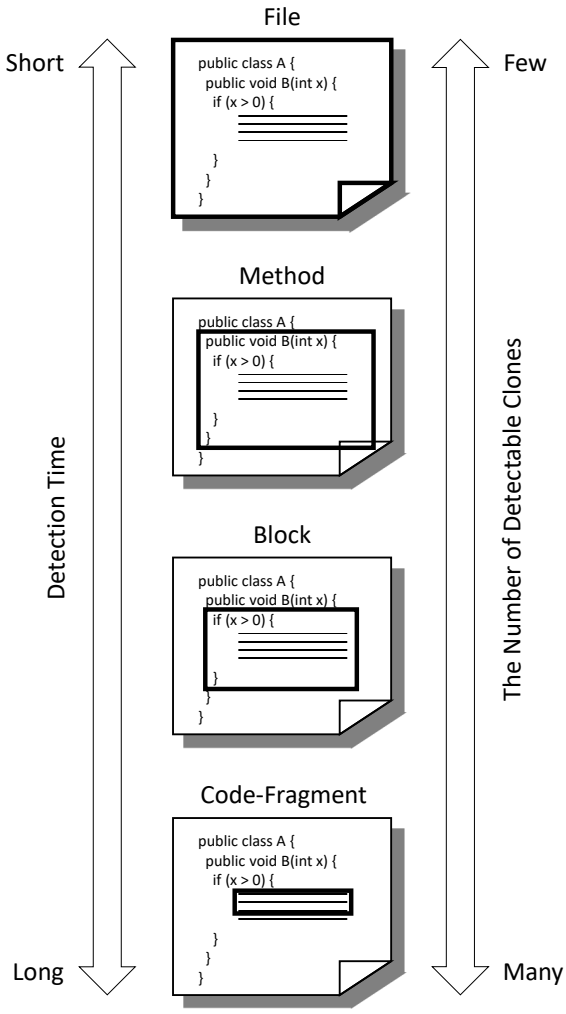


Fig. 1. The Features of Each Detection Techniques

Coarse and fine-grained detection techniques have the following advantages and disadvantages, respectively. The features of those detection techniques are shown in Figure 1.

Detection Time: the smaller the granularity of the detection target is, the longer the detection time. The number of files, methods, and blocks is much fewer than the number of statements, tokens, vertices in tree structures. Therefore, coarse detection techniques can detect clones at a higher speed than fine-grained detection ones.

The number of detectable clones: the larger the granularity of the detection target is, the fewer detectable clones are. Coarse detection techniques cannot detect clones whose code fragments are parts of files, methods, or blocks. Therefore,

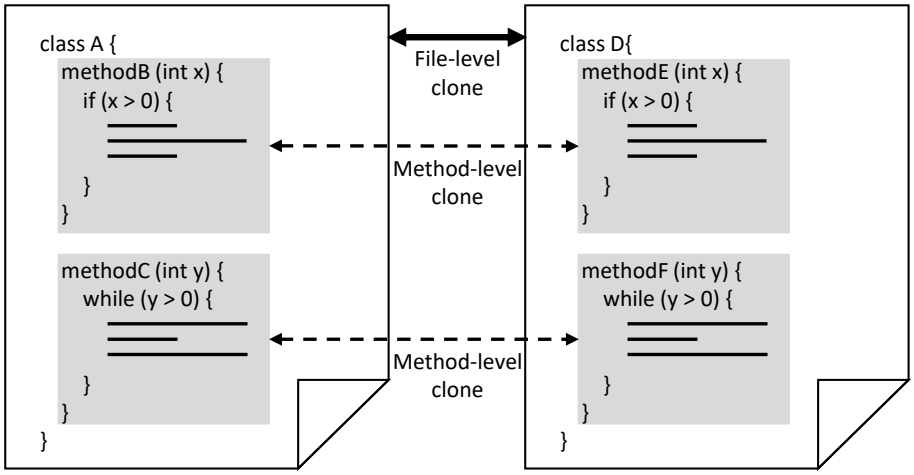


Fig. 3. Secondary Merit of Our Proposed Technique

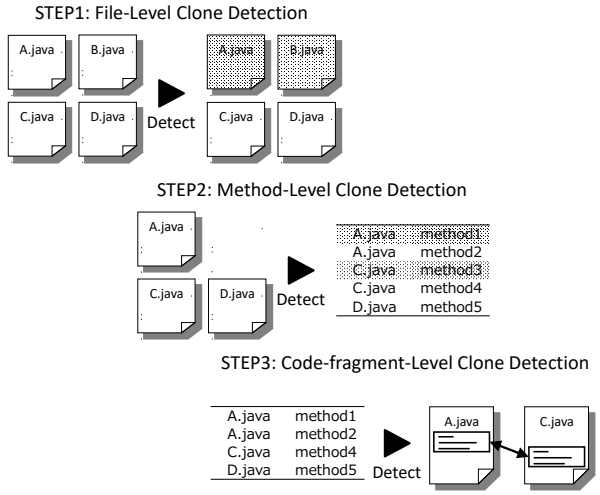


Fig. 2. Overview of Our Proposed Technique

fine-grained detection techniques can detect more clones than coarse detection ones.

In this paper, we propose a technique that detects in order from coarse clones to fine-grained ones (multi-grained detection technique). More concretely, our proposed technique detects clones in the order of file-level, method-level, and code-fragment-level. In the coarse-to-fine-grained-detections, code fragments detected as clones at a certain granularity are excluded from detection targets of more fine-grained detections. Our proposed technique can detect clones faster than fine-grained detection techniques. Besides, it can detect more clones than coarse detection techniques.

In a previous study, equivalence files partitioning is conducted to the input source files prior to the code-fragment-level (token-level) detection [5]. This preprocessing is similar to the file-level clone detection. However, the number of file-level clones is fewer than the number of method-level clones. Excluding method-level clones in addition to file-level clones is more effective to save the detection time. Thus, our proposed technique detects method-level clones in addition to file-level ones and code-fragment-level ones. Moreover, our proposed technique can generate clone detection results which are easier to understand than the technique in the previous study. This is because our proposed technique detects file-level clones, not as preprocessing, and clearly indicates the granularity levels of detected clones.

We have developed a tool based on our proposed technique and applied it to some open source software. Then, we evaluated our proposed technique compared to coarse detection techniques and fine-grained detection ones.

The main contributions of this research are as follows.

- We proposed a technique that detects in order from coarse clones to fine-grained ones.
- We confirmed that our multi-grained detection technique can detect clones faster than fine-grained detection techniques.
- We confirmed that our multi-grained detection technique can detect more clones than coarse detection techniques.
- We confirmed that our multi-grained detection technique can generate clone detection results that are easier to analyze than coarse detection techniques and fine-grained detection ones.

II. Proposed Technique

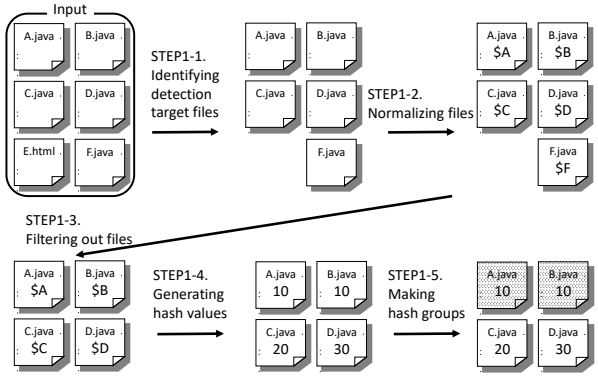
We propose multi-grained detection technique to solve the disadvantages described in Section I.

A. Overview of Our Proposed Technique

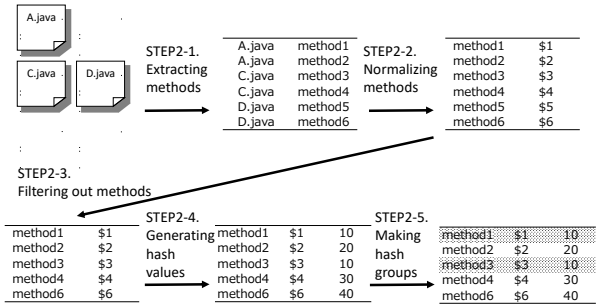
Our proposed technique detects clones in the order of file-level, method-level, and code-fragment-level. An image of our proposed technique is shown in Figure 2.

In the file-level clone detection, files detected as clones are excluded from detection targets of the next method-level clone detection. In the method-level clone detection, methods detected as clones are excluded from detection targets of the next code-fragment-level clone detection.

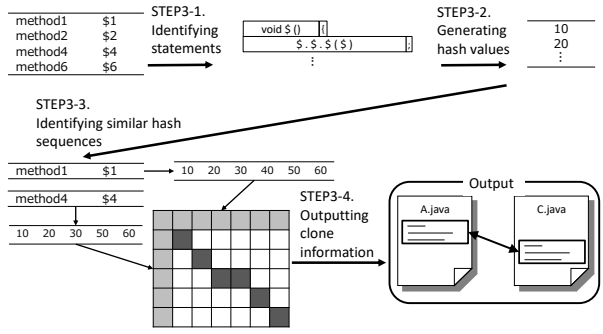
There is a research report that about 49% of methods in about 13,000 software were method-level clones [3]. Since there are a large number of method-level clones across software. The authors expect that the code-fragment-level clone



(a) File-Level Clone Detection



(b) Method-Level Clone Detection



(c) Code-Fragment-Level Clone Detection

Fig. 4. The Details of Each Clone Detection

detection time can be greatly saved by excluding detected methods from detection targets of the next code-fragment-level clone detection.

B. Secondary Merit of Our Proposed Technique

Detecting multiple clone pairs as a single clone pair can reduce the number of detected clones. An example is shown in Figure 3. For example, in the method-level clone detection, two clone pairs of method B and method E, method C and method F are detected. However, it is possible to detect as a single clone pair of class A and class D in the file-level clone detection prior to the method-level clone detection. In other words, by detecting such contiguous clones at a stage with coarser detections, our proposed technique can generate clone detection results which are easier to analyze than fine-grained detection techniques.

III. Implementation

We have developed a tool, **Decrescendo** based on our proposed technique. The inputs of Decrescendo are as follows:

- single or multiple software (currently, Decrescendo targets only source code written in Java),
- minimum clone length (minimum number of tokens considered to be clones), and
- maximum gap rate (ratio of gapped tokens in the detected tokens).

The outputs are clone detection results, such as location information and types. The detection results are output to a database. A simplified procedure of Decrescendo is as follows.

STEP1: File-level clone detection

STEP1-1: identifying detection target files
STEP1-2: normalizing files
STEP1-3: filtering out files
STEP1-4: generating hash values
STEP1-5: making hash groups

STEP2: Method-level clone detection

STEP2-1: extracting methods
STEP2-2: normalizing methods
STEP2-3: filtering out methods
STEP2-4: generating hash values
STEP2-5: making hash groups

STEP3: Code-fragment-level clone detection

STEP3-1: identifying statements
STEP3-2: generating hash values
STEP3-3: identifying similar hash sequences
STEP3-4: outputting clone information

Decrescendo can switch detection on/off for each granularity by setting. Even if the method-level clone detection is off, the inputs of the code-fragment-level clone detection are methods. Besides, even if the file-level and, or the method-level clone detection are off, filterings (STEP1-3, STEP2-3) are not conducted.

The details of each STEP are described below.

A. File-level Clone Detection

The procedure for detecting file-level clones is as follows. An overview is shown in Figure 4(a).

STEP1-1: identifying detection target files

Decrescendo identifies detection target files from single or multiple software given as the inputs. Decrescendo currently analyzes only source code written in Java.

STEP1-2: normalizing files

The following normalization processing is conducted to files identified in STEP1-1.

- Removing white space, tabs, blank lines, annotations, and comments.

- Replacing variables and literals with special tokens.

This process makes it possible to detect two files as clones even if their coding styles differ between them, or their variables and literals are different.

STEP1-3: filtering out files
If the number of tokens included in a normalized file is less than a given minimum clone length, the file is excluded from the detection targets. The reasons for the filtering are as follows:

- saving time required for the matching process in STEP1-5,
- excluding files not to be detected as clones.

STEP1-4: generating hash values
Hash values are generated for each detection target file. Files with the same hash value are a file-level clone. Decrescendo uses MD5 as a hash function.

STEP1-5: making hash groups
Decrescendo groups files whose hash values are the same. If a group consists of two or more files, it is detected as a file-level clone. For example, in Figure 4(a), A.java and B.java are a file-level clone. In the file-level clone detection, Type-1 and Type-2 clones are detected.

B. Method-level Clone Detection

The procedure for detecting method-level clones is as follows. An overview is shown in Figure 4(b).

STEP2-1: extracting methods
The inputs are files which did not have the same hash values in STEP1-5. In addition, Decrescendo randomly selects a file from each group including two or more files whose hash values are the same. Those files are added to the inputs as files representing the groups. For example, in Figure 4(a), A.java and B.java are a file-level clone. In Figure 4(b), A.java is added to the input as a representing file. This is a process for preventing oversight in the method-level clone detection. If the inputs were normalized files, types of method-level clones could not be classified. Thus, the inputs are files before the normalization. Then, Decrescendo builds abstract syntax trees for every file and extract their subtrees corresponding to methods.

STEP2-2: normalizing methods
Decrescendo normalizes methods identified in STEP2-1 as well as STEP1-2.

STEP2-3: filtering out methods
If the number of tokens included in a normalized method is less than a given minimum clone length, it is excluded from the detection targets.

STEP2-4: generating hash values
Hash values are generated for each detection target method. Methods with the same hash value are a method-level clone. Decrescendo uses MD5 as a hash function.

STEP2-5: making hash groups
Decrescendo groups methods whose hash value are the same. If a group consists of two or more methods, it is detected as a method-level clone. For example, in Figure 4(b), method1 of A.java and method3 of C.java are a method-level clone. In the method-level clone detection, Type-1 and Type-2 clones are detected.

C. Code-fragment-level Clone Detection

The procedure for detecting code-fragment-level clones is as follows. An overview is shown in Figure 4(c).

STEP3-1: identifying statements
The inputs are methods which did not have the same hash values in STEP2-5. In addition, Decrescendo randomly selects a method from each group including two or more methods whose hash values are the same. Those methods are added to the inputs as methods representing the groups. For example, in Figure 4(b), method1 of A.java and method3 of C.java are method-level clones. In Figure 4(c), method1 is added to the input as a representing method. Then, Decrescendo identifies statements for these input methods. We define a statement as every subsequence between semicolon, opening brace and closing brace. Decrescendo records the number of tokens included in every statement.

STEP3-2: generating hash values
Hash value is generated for every statement identified in STEP3-1. Decrescendo uses MD5 as a hash function.

STEP3-3: identifying similar hash sequences
We use Smith-Waterman algorithm [6] to identify similar hash sequences. Smith-waterman algorithm is used in the field of biology. This algorithm detects pairs of similar partial alignments from two alignments. It has an advantage that it can identify similar alignments even if they include some gaps. The reason why we uses Smith-Waterman algorithm is that the clone detection tool to which Smith-Waterman algorithm is applied is faster to detect than the other Type-3 clone detection tools [7]. Decrescendo identifies similar hash sequences for each combination of detection target methods.

STEP3-4: outputting clone information
If the following two conditions are satisfied, clone information is output. In the code-fragment-level clone detection, Type-1, Type-2 and Type-3 clones are detected.

- $match \geq \theta$,
- $gap/match \leq \phi$,

where $match$, gap, θ, and ϕ represent the number of tokens included in the similar statements, the number of tokens included in the mismatch statements, a given minimum clone length, and a given maximum gap ratio.

IV. Experiment

We applied Decrescendo to some open software. Then, we evaluated our proposed technique compared to coarse detection technique and fine-grained detection technique. In this experiment, minimum clone length is 50 tokens, and maximum gap rate is 0.3. These are based on previous studies [8] [9].

A. Experimental Questions

EQ1: can the multi-grained detection technique detect clones faster than fine-grained detection technique?

TABLE II
EXPERIMENTAL RESULTS

Detection Granularity			# of Detected Clone Pairs				Detection Time [s]
File	Method	Code-fragment	File	Method	Code-fragment	Sum	
✓			1,067	-	-	1,067	4.0
	✓		-	31,807	-	31,807	15.3
		✓	-	-	202,537	202,537	4,695.4
✓	✓		1,067	31,291	-	32,358	15.9
✓		✓	1,067	-	202,021	203,088	4,137.7
	✓	✓	-	31,807	170,730	202,537	690.2
✓	✓	✓	1,067	31,291	170,730	203,088	671.6

EQ2: can the multi-grained detection technique detect more clones than coarse detection technique?

EQ3: are multi-grained detection results easier to analyze than coarse detection results and fine-grained detection ones?

To research these questions, we executed Decrescendo in all cases where detection on each granularity was switched on/off.

B. Experimental Environment

The CPU of the computer used in this experiment is 2.40 GHz Intel Xeon CPU (8 logic processors), and the memory size is 32.0 GB. Experimental targets and output database were located on SSD.

C. Experimental Target

Table I shows an overview of target software. We randomly selected 10 latest software from Apache's repository[1] as of September 13, 2016. To avoid clone detection from the same software with different versions, only files under trunk are detection target.

D. Experimental Results

Table II shows experiment results. Columns of Detection Granularity indicate whether the detection is on or not. We answer to each EQ with Table II.

1) EQ1: comparing the case of detecting all granularities clones and the case of detecting only code-fragment-level clones, the detection time is greatly saved (4,695.4s→671.6s). The time required for each STEP is shown in Table III. The execution time from STEP2-1 to STEP2-4 is slightly saved (13.4s→10.1s). This is because detected file-level clones are excluded from the inputs of the method-level clone detection.

Furthermore, the most remarkable point is that the execution time of STEP3-3 was greatly saved (4,672.0s→637.0s). In detecting all granularities clones, the execution time is about 1/7 against the code-fragment clone detection. In both cases, almost all of the total execution time is the execution time of STEP3-3. Thus, excluding detected files and methods, and also, files and methods which are less than a given minimum clone length is effective for saving the detection time.

Moreover, comparing the case of detecting file-level, code-fragment-level clones and the case of detecting all granularities clones, excluding detected methods, and also, methods which are less than a given minimum clone length is especially effective for saving the detection time.

In conclusion, our answer to EQ1 is **YES**. The multi-grained detection technique can detect clones greatly faster than the fine-grained detection techniques. A large number of file-level and method-level clones were detected in cases of detecting clones from a large set of source code in previous studies [3] [10] [11]. Especially in such cases, the authors consider that the multi-grained detection technique can save the detection time effectively.

2) EQ2: comparing the case of detecting file-level, method-level clones and the case of detecting all granularities clones, the number of detected clone pairs increased (32,358→203,088). Comparing the case of detecting code-fragment-level clones and the case of detecting all granularities clones, the number of detected clone pairs did not differ much (202,537→203,088).

In conclusion, our answer to EQ2 is **YES**. The multi-grained detection technique can detect more clones than the coarse detection techniques. Moreover, multi-grained detection

TABLE I
OVERVIEW OF TARGET SOFTWARE

software	# of Java files	LOC
Any23	369	46,957
cTAKES	1253	209,545
Forrest	252	32,491
JSPWiki	491	107,530
jUDDI	938	163,103
Onami	572	43,784
OODT	1,628	223,846
OpenOffice	3871	774,670
Roller	612	96,617
Wink	1,372	210,167
Sum	11,358	1,908,710

[1]http://svn.apache.org/repos/asf/

TABLE III
THE TIME REQUIRED FOR EACH PROCESS

STEP	All [s]	Code-fragment [s]
STEP1-1_STEP1-4	3.7	4.0
STEP1-5	0.0	-
Output(File)	2.0	-
STEP2-1_STEP2-4	10.1	13.4
STEP2-5	0.0	-
Output(Method)	1.4	-
STEP3-1_STEP3-2	1.6	1.5
STEP3-3	637.0	4,672.0
Output(Code-fragment)	13.8	3.8

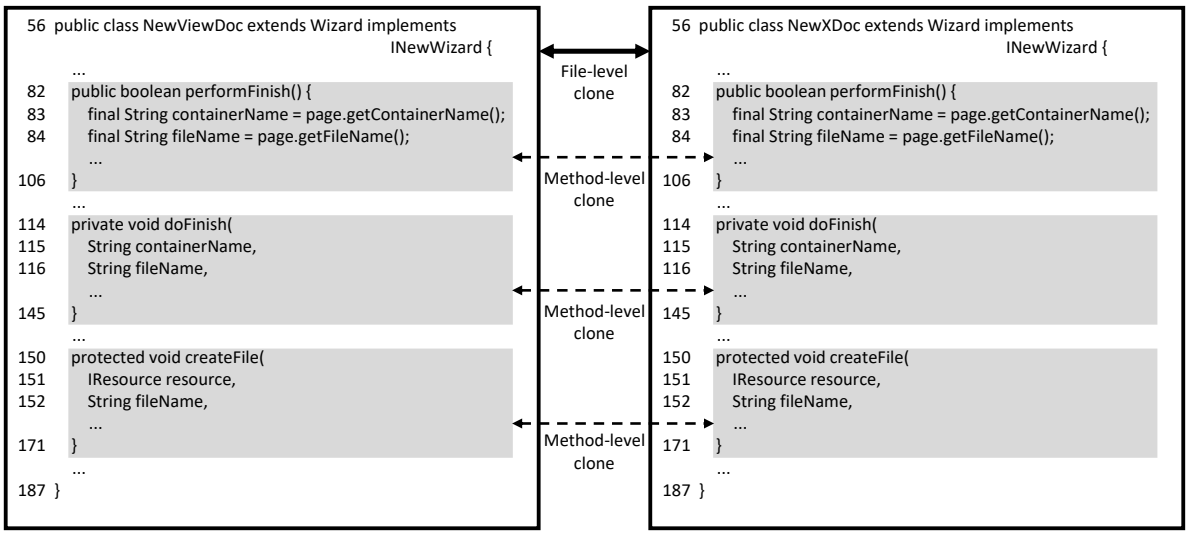


Fig. 5. An Example where Multiple Method-level Clones are Detected as a File-level Clone

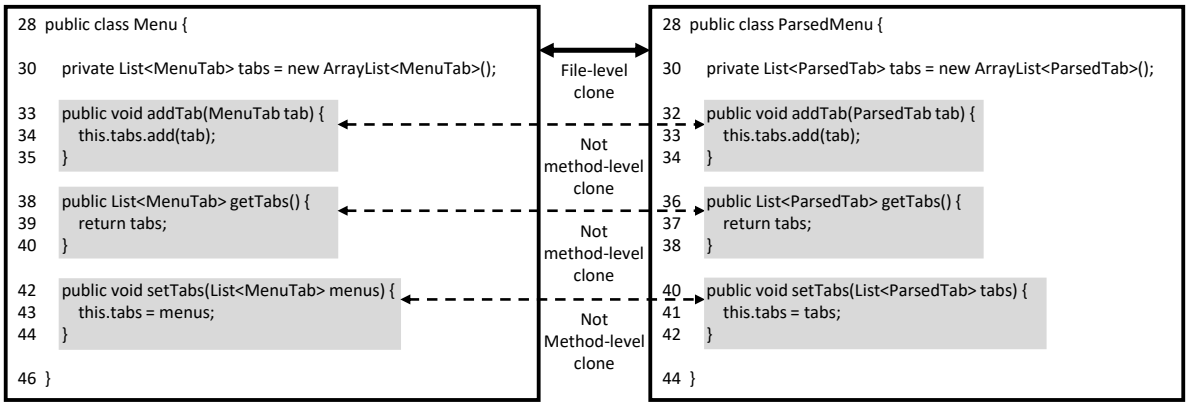


Fig. 6. An Example where a File-level Clone can be Refactored

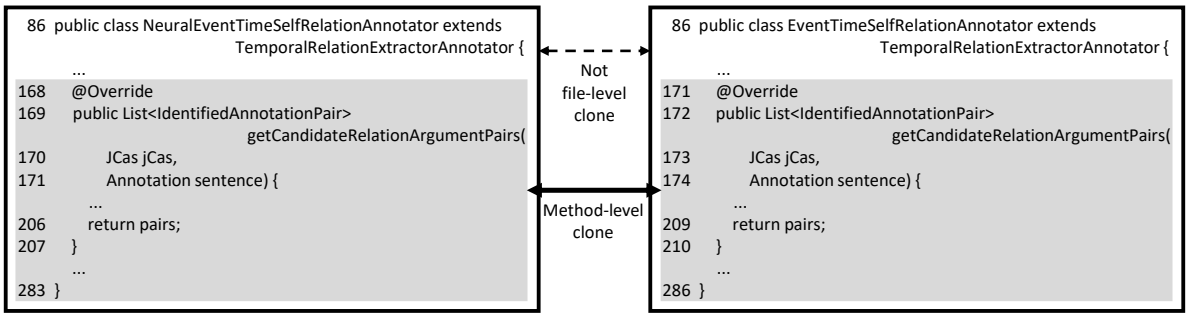


Fig. 7. An Example where a Method-level Clone can be Refactored

technique detects as many clones as fine-grained detection techniques. This indicates that the detection time has saved without losing detected clones.

3) EQ3: we confirmed three cases generating clone detection results which were easy to analyze.

The first case is that multiple method-level clone pairs were detected as a file-level clone pair. An example is shown in Figure 5. When the file-level clone detection is off, *performFinish*, *doFinish*, and *createFile* methods were detected as different method-level clone pairs. However, when the file-level clone detection is on, they were detected as a single clone pair of *NewXDoc* and *NewViewDoc* classes. In the file-level clone detection, 516 method-level clone pairs were detected as 404 file-level clone pairs. By detecting such contiguous clones at a stage with coarser detections, there is a possibility that we notice refactoring opportunities to merge multiple classes into a class. In this case, the multi-grained detection technique can generate clone detection results which are easier to analyze than fine-grained detection techniques.

The second case is that methods in a file were not detected in the method-level clone detection although the file was detected in the file-level clone detection. This is because

all methods in the file-level clone were less than a given minimum clone length. It is not necessary to detect methods as method-level clones, but there are cases where the entire class needs to be detected as a file-level clone. An example is shown in Figure 6. When the file-level clone detection is off, *addTab*, *getTabs*, and *setTabs* methods were not detected because those methods were less than a given minimum clone length. However, when the file-level clone detection is on, *Menu* and *ParsedMenu* classes were detected as a file-level clone because those classes were more than a given minimum clone length. By detecting such file-level clones, there is a possibility of refactoring opportunities. In this case, the multi-grained detection technique can also generate clone detection results which are easy to analyze.

The third case is that methods in files which were not detected as file-level clones were detected as method-level clones. An example is shown in Figure 7. In the method-level detection, *getCandidateRelationArgumentPairs* methods in *NeuralEventTimeSelfRelationAnnotator* and *EventTimeSelfRelationAnnotator* classes were detected as method-level clones. In addition, *NeuralEventTimeSelfRelationAnnotator* and *EventTimeSelfRelationAnnotator* classes were extended the same *TemporalRelationExtractorAnnotator* class, *getCandidateRelationArgumentPairs* methods were overridden. Those methods can be pulled up to the parent class. The multi-grained detection technique generates clone detection results which include granularity levels of detected clones, and from above examples, each granularity clone is refactored in different ways. Thus, by using the information of granularity levels, its detection results become easy to analyze because refactoring techniques to be applied become clearer.

In conclusion, our answer to EQ3 is **YES**.

V. Thread to Validity

Target systems: we targeted only 10 software. If we had selected other software, the results might have been different from this experiment. However, even if we select other software, we consider that we can obtain the same experimental results.

Hash collision: we use MD5 as a hash function. If hash collisions occur, non-duplicated files, methods, and code fragments are accidentally regarded as clones. However, in this research, MD5 which outputs a 128-bit hash value is used, and the probability of a hash collision is considered to be sufficiently low.

Normalization: in this experiment, the normalization described in Section III was conducted. If we had conducted a different normalization, the clone detection results would have been different from this experiment.

VI. Conclusion

We proposed a technique that detects in order from coarse clones to fine-grained ones (multi-grained detection technique). We have developed a tool based on our proposed technique and applied it to some open source software. Then, we evaluated our proposed technique compared to coarse detection techniques and fine-grained detection techniques.

The main contributions of this research are as follows.

- We proposed a technique that detects in order from coarse clones to fine-grained ones.
- We confirmed that our multi-grained detection technique can detect clones faster than fine-grained detection techniques.
- We confirmed that our multi-grained detection technique can detect more clones than coarse detection techniques.
- We confirmed that our multi-grained detection technique can generate clone detection results that are easier to analyze than coarse detection techniques and fine-grained detection ones.

Our future works are as follows.

- Supporting for other languages.
- Comparing with other clone detection tools.
- Evaluating the multi-grained detection technique with traditional quality metrics such as precision, recall, and f-measure.
- Detecting clones from larger set of source code to find library candidates or overlooked bugs.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Number JP25220003.

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] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Inter-project functional clone detection toward building libraries-an empirical study on 13,000 projects," in *Proc. of the 19th Working Conference on Reverse Engineering*, 2012, pp. 387–391.

[4] K. Hotta, J. Yang, Y. Higo, and S. Kusumoto, "How accurate is coarse-grained clone detection?: Comparison with fine-grained detectors," in *Proc. of the 8th International Workshop on Software Clones*, 2014, pp. 1–18.

[5] E. Choi, N. Yoshida, Y. Higo, and K. Inoue, "Proposing and evaluating clone detection approaches with preprocessing input source files," *IEICE Transactions on Information and Systems*, vol. 98, no. 2, pp. 325–333, 2015.

[6] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[7] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Gapped code clone detection with lightweight source code analysis," in *Proc. of the 21st International Conference on Program Comprehension*, 2013, pp. 93–102.

[8] 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.

[9] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *Proc. of the 31st International Conference on Software Maintenance and Evolution*, 2015, pp. 131–140.

[10] Y. Sasaki, T. Yamamoto, Y. Hayase, and K. Inoue, "Finding file clones in freebsd ports collection," in *Proc. of the 7th Working Conference on Mining Software Repositories*, 2010, pp. 102–105.

[11] J. Ossher, H. Sajnani, and C. Lopes, "File cloning in open source java projects: The good, the bad, and the ugly," in *Proc. of the 27th International Conference on Software Maintenance*, 2011, pp. 283–292.