2H-06 A Code Clone Detection Technique for Object-Oriented Programming Languages and Its Empirical Evaluation

Toshihiro Kamiya†, Shinji Kusumoto†, and Katsuro Inoue†‡
†Graduate School of Engineering Science, Osaka University
‡Graduate School of Information Science, Nara Institute of Science and Technology
{kamiya, kusumoto, inoue}@ics.es.osaka-u.ac.jp

1 Introduction

A code clone is a code portion in source files that is identical or similar to another. Clones are introduced because of various reasons such as reusing code by 'cut-and-paste' or intentionally repeating a code portion for performance enhancement[2]. Clones make the source files very hard to modify consistently. For example, assume that a software system has several clone subsystems created by duplication with slight modification. When a fault is found in one subsystem, the engineer has to carefully modify all other subsystems. For a large and complex system, there are many engineers who take care of each subsystem, and modification becomes very difficult. Various clone detection tools have been proposed and implemented [1][2][5][6][7], and a number of algorithms for finding clones have been used for them, such as line-by-line matching for an abstracted source program [1], and similarity detection for metrics values of function bodies [7].

1.1 Definition of clone and related terms

A **clone-relation** is defined as an equivalence relation (i.e., reflexive, transitive, and symmetric relation) on code portions. A clone-relation holds between two code portions if (and only if) they are the same sequences. For a given clone-relation, a pair of code portions is called **clone-pair** if the clone-relation holds between the portions. An equivalence class of clone-relation is called **clone-class**. That is, a clone-class is a maximal set of code-portions in which a clone-relation holds between any pair of code-portions.

For example, suppose a file has the following 12 tokens:

a x y z b x y z c x y d

We get the following three clone-classes:

C1) $a\ \underline{x\ y\ z}\ b\ \underline{x\ y\ z}\ c\ x\ y\ d$

C2) $a\ \underline{x\ y}\ z\ b\ \underline{x\ y}\ z\ c\ \underline{x\ y}\ d$

C3) $a\ x\ \underline{y\ z}\ b\ x\ \underline{y\ z}\ c\ x\ y\ d$

Note that sub-portions of code portions in each clone-class also make clone-classes (e.g. Each of C3 is a sub-portion of C1). In this paper, however we are interested only in maximal portions of clone-classes so only the latter are discussed.

2 Proposed clone-code detection technique

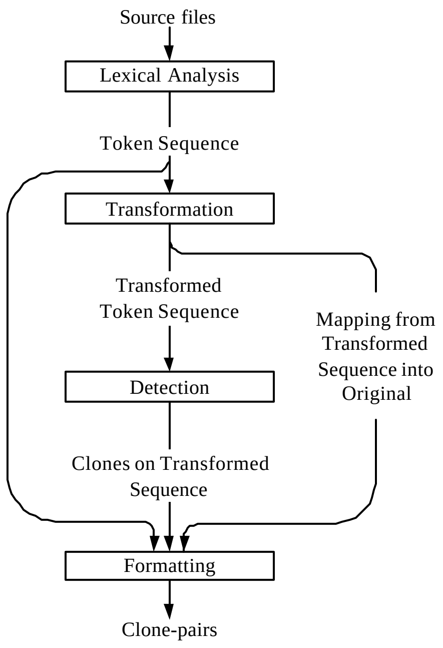
Our approach presented in this paper concerns the following issues in clone detection.

- **Identification of structures**

Our pilot experiment has revealed that certain types of clones seem difficult to be rewritten as a shared code even if they are found as clones. Examples are a code portion that begins at the middle of a function definition and ends at the middle of another function definition, and a code portion that is a part of a table initialization code. For effective clone analysis, our clone detection technique automatically identifies and separates each function definition and each table definition code. For comparison, in [1], table initialization values have to be removed by hand, whereas in [7], only an entire function definition can become a candidate for clone.

- **Regularization of identifiers**

Recent programming languages such as C++ and Java provide *name space* and/or *generic type*. As a result, identifiers often appear with attributive identifiers of name space and/or template arguments. In order to treat each complex name as an equivalent simple name, the clone detecting process has a subprocess to transform complex names into simple form. If source files are represented as a string of tokens, structures in source files (such as sentences or function definitions) are represented as substrings of tokens, and they can be compared token-by-token to identify clones. Identifying structures and transforming names require knowledge of syntax rules of the programming languages. Therefore, the implementation of the clone detecting technique depends on the input. The detail of clone detecting process is described in Section 2.1.



Source files
Lexical Analysis
Token Sequence
Transformation
Transformed Token Sequence
Mapping from Transformed Sequence into Original
Detection
Clones on Transformed Sequence
Formatting
Clone-pairs

Figure 1. Clone detecting process

2.1 Clone-detecting process

Clone detecting is a process in which the input is source files and the output is clone-pairs. The entire process of our token-based clone detecting technique is shown in Figure 1. The process consists of four steps:

(1)Lexical analysis

Each line of source files is divided into tokens corresponding to a lexical rule of the programming language. The tokens of all source files are concatenated into a single token sequence, so that finding clones in multiple files is performed in the same way as single file analysis. At this step, the white spaces between tokens are removed from the token sequence, but the spaces are sent to the formatting step to reconstruct the original source files.

(2)Transformation

The token sequence is transformed by subprocesses (2-1) and (2-2) described below. At the same time, the mapping information from the transformed token sequence into the original token sequences is stored for the later formatting step.

(2-1)Transformation by the transformation rules

The token sequence is transformed, i.e., tokens are added, removed, or changed based on the transformation rules. Table 1 shows the transformation rules for Java source code(For C++ source code, another transformation rules are adapted).

(2-2)Parameter replacement

After step 2-1 each identifier related to types, variables, and constants is replaced with a special token (this replacement is a preprocess of the 'parameterized match' proposed in [1]). This replacement makes code-portions in which variables are renamed to be equivalent token sequences.

(3)Detection

From all the substrings on the transformed token sequence, equivalent pairs are detected as clone-pairs. Each clone-pair is represented as a quadruplet (*cp, cl, op, ol*), where *cp and op are, respectively,* the position of the first and second portion, and cl and ol are their respective lengths.

(4)Formatting

Each location of clone-pair is converted into line numbers on the original source files.

Here, a clone-relation is specified with the transformation rules and the parameter-replacement described above. Other clone-relations are derived with a subset of the transformation rules and neglection of the parameter-replacement. In the experiments described in Section 3, a clone-relation with all the transformation rules is compared to a clone-relation with a subset of the transformation rules.

2.2 The implementation techniques of tool CCFinder

Tool CCFinder was implemented in C++ and runs under Windows 95/NT 4.0 or later. CCFinder extracts clone-pairs from C, C++ and Java source files. The tool receives the paths of source files from the command-line (or text files in which the paths are listed), and writes the locations of the extracted clone-pairs to the standard output. The straightforward clone-detecting algorithm for *n* tokens with matrix requires the time complexity of $O(n^2)$. A data structure called

Table 1. Transformation rules for Java

#	Rule
RJ1	(PackageName '.')+ ClassName → ClassName Here, PackageName is a word that begins with a small letter and ClassName is a capitalized word.
RJ2	NDotOrNew NClassName '(' → NDotOrNew CalleeID '.' NClassName '(' Here, NDotOrNew is a token except '.' or 'new'. NClassName is an uncapitalized word. CalleeID is a token for an omitted callee.
RJ3	'=' '{' InitalizationList, '}' → '=' '{' UniqueID '}' ']' '{' InitalizationList, '}' → ']' '{' UniqueID '}' Here, InitalizationList is a sequence of Name, Number, String, Operators, ',', '(', ')', '{', and '}'.
RJ4	Insert UniqueID at each end of the top-level definitions and declaration.

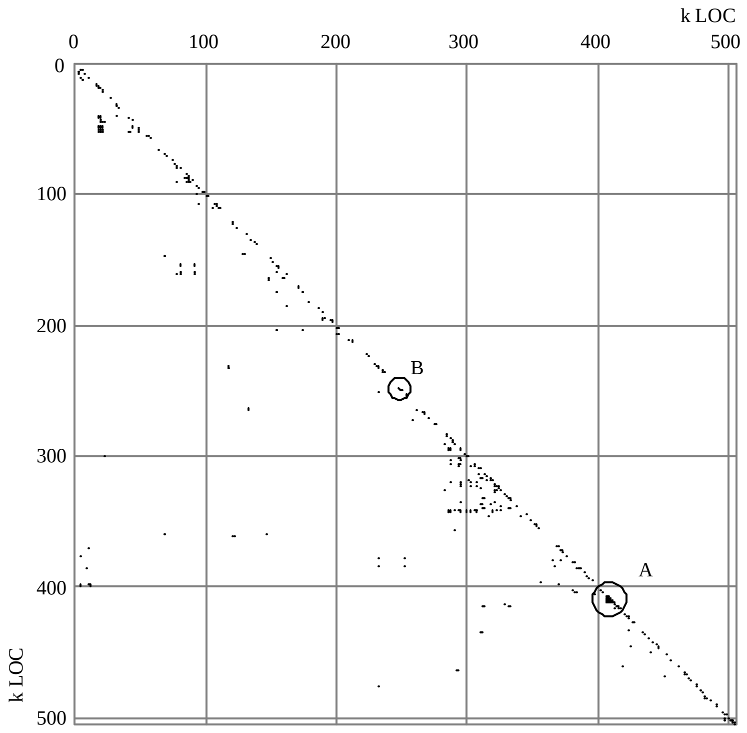


Figure 2. Scatter plot of clones over 20 lines in JDK

suffix-tree is devised to detect clone-pairs and it requires $O(n)$ time complexity[4]. CCFinder employs a relaxed algorithm of $O(n \log n)$ time using a suffix-tree, which is not only easily implemented but also practically efficient.

As the other optimization for large source files, CCFinder uses a filtering for tokens: The clone-detection algorithm distinguishes "header" tokens. A header token is defined as the token that can be the first token of code portions of code-pairs. For example, on detecting clone-pairs in C/C++ source files, tokens, "#", "{", and "(" are header tokens by themselves. Also, the successors of ":", "; ", ")", "}", and ends-of-line of a preprocessor directive become header tokens. This filtering reduced the number of tokens inserted into suffix-tree by factor 3 in either C/C++ or Java source file, in the experiments described in Section 4.

3 Experiment

The purpose of the experiment was to evaluate our token-based clone-detecting technique and the metrics. The target source files have 'industrial' size and are widely available. The person who performed the analysis did not have preliminary knowledge about the source files; consequently the following results are obtained purely by the analysis with the tool and metrics. In all the following experiments, tool CCFinder was executed on a PC with Pentium III 650MHz and 1GB RAM.

3.1 4.1 Clones in a Java library, JDK

JDK 1.2.2 is a commonly used Java library and the source files are publicly

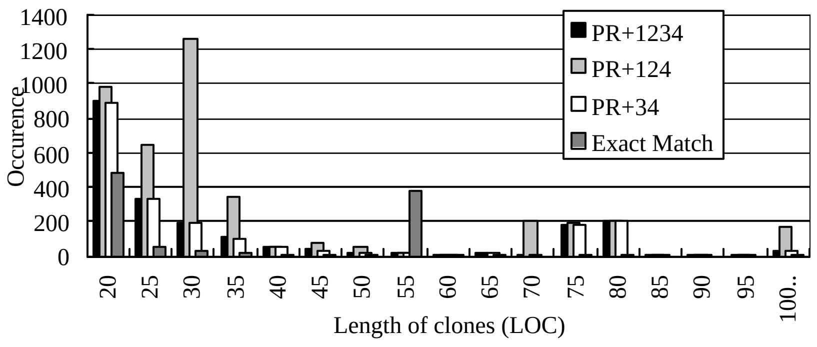


Figure 3. Occurences against length of clone-pairs in JDK

available. Tool CCFinder has been applied to all source files of JDK excluding examples and demo programs, which are about 500k lines in total, in 1648 files. It takes about 3 minutes for execution on the PC. Figure 2 shows a scatter plot of the clone-pairs having at least 20 lines of code (LOC). Both the vertical and horizontal axes represent lines of source files. The files are sorted in alphabetical order of the file paths, so files in the same directory are also located near on the axis. A clone-pair is shown as a diagonal line segment. Only lines below the main diagonal are plotted as mentioned in Section 2.1. In Figure 2, each line segment looks like a dot because each clone-pair is small (several decades lines) in comparison to the scale of the axis. Most line segments are located near the main diagonal line, and this means that most of the clones occur within a file or among source files at the near directories.

Crowded clones marked *A* in the graph correspond to 29 files of `javax/ swing/ plaf/ multi/ *.java`. These files are very similar to each other and some of them contain an identical class definition except for their different parent classes. According to the comments of the source files, a code generator named `AutoMulti` creates the files. To modify these files, the developer should obtain the tool (the tool is not included in JDK), edit, and apply it correctly. If the developer does not use the tool, he/she has to update all the files carefully by hand. As the example shows, the modification of clones needs extra work. In this case, these clones are easily rewritten with a shared code if the programming language would support *generic types.*

The longest clone (349 lines) is found within `java/ util/ Arrays.java` (marked *B* in Figure 2). Methods named "`sort`" have 18 variations for signatures (number and types of arguments), and they use identical algorithm/routine for sorting.

3.2 Evaluation of transformation rules for JDK

In Section 2.1, we also proposed the transformation rules for Java. To evaluate effectiveness of the transformation rules, we have applied CCFinder with some of their transformation rules disabled. Figure 3 shows the histogram of detected clone-pairs when some of rules are applied. PR+1234 means that the parameter-replacement and all rules (RJ1, RJ2, RJ3, and RJ4) are applied (i.e. original CCFinder). Exact Match means that no parameter-replacement or no transformation is applied. This figure shows that the longer the clone length is, the smaller its occurrence becomes. A noticeable peak around 80 LOC is a set of clone-pairs found in files generated by `AutoMulti`, which cannot be detected by Exact Match by the reason mentioned above. In this experiment, the clone-pairs found by PR+1234 are much fewer than with PR+124. This means that rule RJ3 removes many table initialization codes.

The case PR+1234 extracted 2111 clone-pairs and PR+34 extracted 2093 clone-pairs. There are several clone-pairs that can be detected by introducing RJ1 and RJ2. In the case of Exact Match, only a small number of clone-pairs are found. The "exact" clone-pairs are obvious candidates to be rewritten as a shared code. However, our transformation and parameter replacement approach finds more subtle clone-pairs so that the chances to rewrite and reorganize overall structures of software systems become higher.

4 Conclusion

In this paper, we presented a clone detecting technique with transformation rules and a token-based comparison. The technique is implemented and applied to a library of JDK in the experiment, and successfully extracted code-clones.

References

[1] B. S. Baker, "On finding Duplication and Near-Duplication in Large Software System", *Proc. IEEE WCRE '95.*, pp. 86-95 Jul. 1995

[2] I. D. Baxter et. al. "Clone Detection Using Abstract Syntax Trees", *Proc. ICSM '98*, pp. 368-377, Bethesda, Maryland, Nov. 1998.

[3] S. Ducasse et. al. "A Language Independent Approach for Detecting Duplicated Code", *Proc. IEEE ICSM '99*, pp. 109-118. Oxford, England. Aug. 1999.

[4] D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, pp. 89-180. Cambridge University Press 1997.

[5] J. H. Johnson, "Identifying Redundancy in Source Code using Fingerprints", *Proc. of IBM CAS CON '93*, pp. 171-183, Toronto, Ontario. Oct. 1993.

[6] B. Laguë et. al "Assessing the Benefits of Incorporating Function Clone Detection in a Development Process", *Proc. IEEE ICSM '97*, pp. 314-321, Bari, Italy. Oct. 1997.

[7] J. Mayland et. al. "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics", *Proc. IEEE ICSM '96*, pp. 244-253, Monterey, California, Nov. 1996.