社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.
SS2000-47 (2001-01)

コードクローン検出における新手法の提案および評価実験

神谷 年洋[†], 楠本 真二[†], 井上 克郎[†‡]

†大阪大学大学院基礎工学研究科
〒560-8531 大阪府豊中市待兼山町 1-3
‡奈良先端科学技術大学院大学情報科学研究科
〒630-0101 奈良県生駒市高山町 8916－5

{kamiya, kusumoto, inoue}@ics.es.osaka-u.ac.jp

あらまし　コードクローンとは，ソースファイル中の，まったく同じあるいは類似したソースコード断片のことである．クローンは「カット&ペースト」によるコードの再利用や，実行時の性能を向上させるための意図的な繰り返しなど，さまざまな理由で作られる．クローンはソースファイルの首尾一貫した変更を困難にする．オブジェクト指向プログラミング言語で記述されたソースコードから，より正確にクローンを検出するための手法を提案する．提案した手法はツールに実装され，実験により，JDK のソースコードからクローンを抽出することができた．さらに，従来の検出方法では見逃されてしまうようなクローンも発見できた．

キーワード　コードクローン, 重複コード, CASE ツール, メトリクス, 保守

A Token-based Code Clone Detection Technique and Its 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

Abstract　A code clone is a code portion in source files that is identical or similar to another. Since code clones generally reduce maintainability of software, several code clone detection techniques and tools have been proposed. This paper proposes a new clone detection technique, which consists of transformation of input source text and token-by-token comparison. Based on the proposed code clone detection technique, we developed a tool named CCFinder, which extracts code clones in C/C++ or Java source files. As well metrics for code clones were developed. In order to evaluate the usefulness of the tool and metrics, we conducted several experiments. As the results, the tool found several subsystems in two operating systems, namely FreeBSD and Linux, that could be traced to the same original. As well, the proposed metrics found interesting clones in a Java library, JDK.

Code clone, Duplicated code, CASE tool, Metrics, Maintenance

key words

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]0[6][7][8], 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 [8].

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[1]. 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.

[1] Sequences are sometimes original character strings, strings without white spaces, sequences of token type, and transformed token sequences. We will discusses how we deal with such sequences.

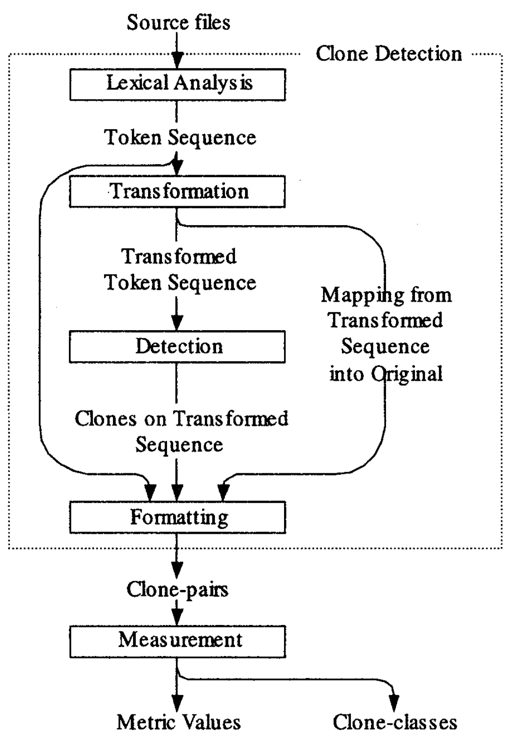

Figure 1. Clone detecting process

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 [8], 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* [3]. 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.

- **Ranking clones by importance**

Large software systems often include many clones, so a clone analysis method must distinguish important clones from many 'uninteresting' clones. The metrics presented in Section 3.3 enable to identify such important clones: clones that enable large code reduction by their removal, or clones that have so widely spread in the system that are difficult to find by hand and to maintain. A certain metric value is used to estimate how many lines of source files are reduced by making a shared routine of each clone, and another is used to evaluate how each clone is spread over a software system.

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

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.

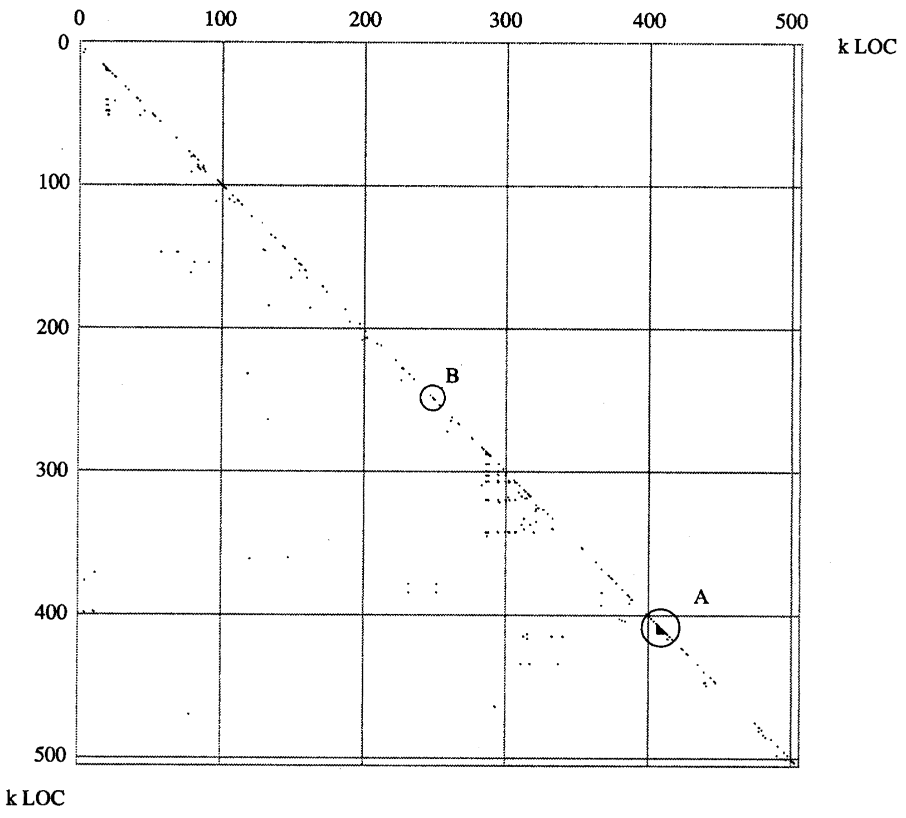

Figure 2. Scatter plot of clones over 20 lines in JDK

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 suffix-tree is devised to detect clone-pairs and it requires $O(n)$ time complexity[1][5]. 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.

The optimizations employed by CCFinder for large source files are the following:

- **Filtering by header tokens**

We would like to extract the code portions that make real sense as a clone-pair. As a simple filtering for this purpose, 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.

- **Integer token**

A token is represented by a serial number, not as a string or a hash-value. This optimization is enabled by parameter-replacement, which causes a token sequence to consist of only limited kinds of tokens. Otherwise, a set of tokens is infinite in general, thus the tool should use string or hash-value as a representation of a token, which would cost higher time and space in clone detection.

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.

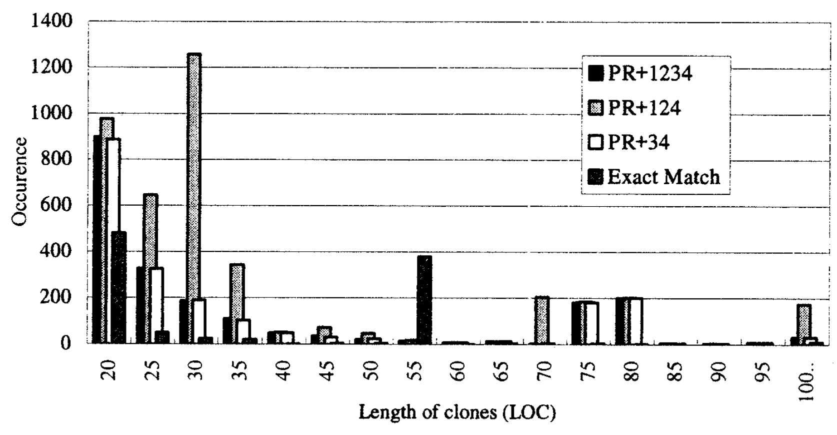

Figure 4. Occurences against length of clone-pairs in JDK

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 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.

Figure 3 shows a part of a file `MultiButtonUI.java`. This file contains same to a file `MultiColorChooserUI.java`, except lines 32, 161, and 163. 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 type* [3].

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 4 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

```
 31|  */
 32|public class MultiButtonUI extends ButtonUI {
 33|

160|   public static ComponentUI createUI(JComponent a) {
161|      ComponentUI mui = new MultiButtonUI();
162|      return MultiLookAndFeel.createUIs(mui,
163|             ((MultiButtonUI) mui).uis,
164|             a);
165|   }
```

Figure 3. A clone file MuitiButtonUI.java found in JDK

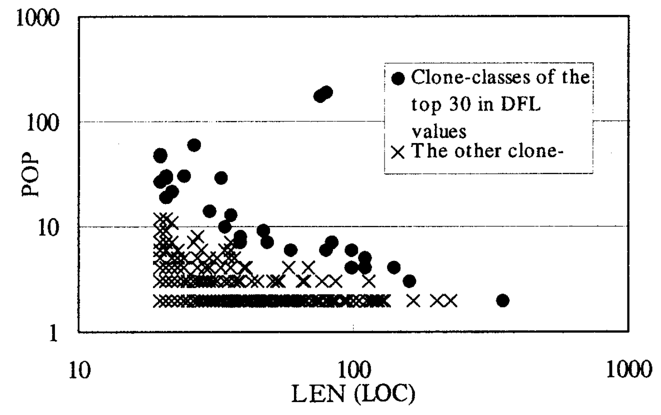

Figure 5. Population and length of clone-classes in JDK

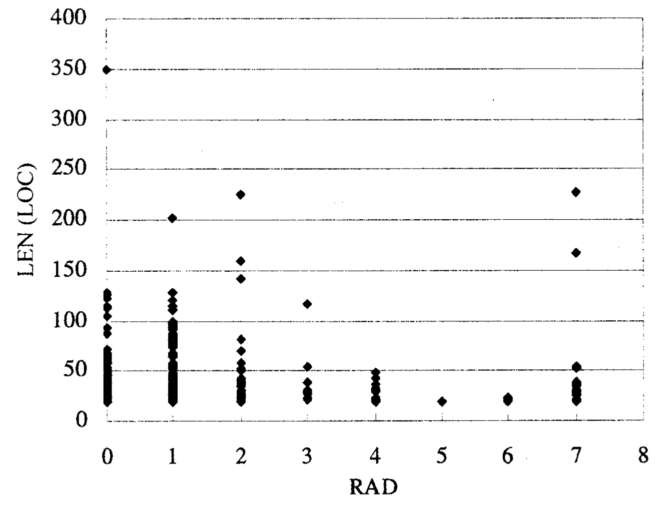

Figure 6. Length and Radius of clone-classes in JDK

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.

3.3 Analysis using clone metrics

We define several metrics for clone-classes in order to find important clone-classes, which enable us to perform large code reduction. Also, we use metrics to find clone-classes that are widely spread over a system.

Radius of clone-class; *RAD*(*C*)

For a given clone-class *C*, let *F* is a set of files which include each code portion of *C*. Define *RAD*(*C*) as the maximum length of path from each file *F* to the lowest common ancestor directory of all files in *F*. For example, a clone class *C* contains two code portions and one of them exists in 'abc/def/ghi/source1.c' and the other in a file 'abc/def/xyz/source2.c', then RAD(*C*), i.e. the length of the common path 'abc/def/', is equal to 2. If all code portions of *C* are included in one file, define *RAD*(*C*) = 0.

If a clone-class has a large RAD, the code portions widely spread over a software system, and it would become difficult to find those clones and maintain their consistency correctly, since such different subsystems are likely to be maintained by different engineers.

Length; *LEN*(*C*), *LEN*(*p*)

LEN(*p*) is the number of lines of a code portion *p*. *LEN*(*C*) for clone-class *C* is the maximum *LEN*(*p*) for each *p* in *C*.

Population of clone-class; *POP*(*C*)

POP(*C*) is the number of elements of a given clone-class *C*.

A clone class with a large POP means that similar code portions appear in many places.

Deflation by clone-class; *DFL*(*C*)

Combination of LEN and POP gives an estimation of how many lines would be removed from source files by rewriting each clone-class as a shared code. Suppose that all code-portions of a clone-class *C* are replaced with caller statements of a new identical routine (function, method, template function, or so) and that this caller statement is one line. In this case *LEN*(*C*) × *POP*(*C*) lines of code are occupied in the original source files. In the newly restructured source files, they occupy *POP*(*C*) lines for caller statements and *LEN*(*T*) for a callee routine. Now let us define a metric DFL[2] as a rough estimator of reduced source lines:

$$\begin{aligned} DFL(C) &= (\text{old LOC of } C) - (\text{new LOC of } C)\\ &= LEN(C) \times POP(C) - (POP(C) + LEN(C))\\ &= (LEN(C) - 1) \times (POP(C) - 1) - 1, \end{aligned}$$

Note that *DFL*(*C*) >= 0, for all clone-classes *C* that

[2] A similar metric is used in [1], which estimates how many lines are removed in total by rewriting all clone-classes.

satisfy *LEN*(*C*) >= 2 and *POP*(*C*) >= 2.

4 Applying Metrics to JDK

The data of JDK were analyzed using the metrics.

Figure 5 shows the LEN and POP parameters of each clone-class. The set of clone-classes with the highest 30 DFL values is obviously different from the set with the highest LEN values or the set with the highest POP values. By investigation of source files, the clone-classes of the top 30 DFL values are classified into the following four types:

- Source files generated by `AutoMulti` (10 clone-pairs)
- Part of a switch/case statement which seems to be easily rewritten by an array (3 clone-pairs)
- Routines to apply one algorithm to many data types, that could be rewritten by generic type (5 clone-pairs)
- Instantiations of definitional computations (e.g. methods in order to put or get a value of an instance value and methods in order to change signature or private/public accessibility of the other methods) (12 clone-pairs)

Figure 6 shows the RAD and LEN parameters of each clone-class. Except for clone-classes whose RAD values are 7, most clone-classes with high LEN have small RAD value. That is, in most cases, a clone occurs between files at near directories. One of the reasons would be that copying a code portion from a distant file is a time consuming job because developer needs to search for the target code portion through many files. Another reason would be that the nearer files are more likely to implement similar functionalities.

As for all clone-classes whose RAD values are 7, 6, or 5, we investigated all the corresponding source files. All code portions of 7 are found in '`swing`' subsystem, which has source files located at distant directories, `com/ sun/ java/ swing` and `javax/ swing`. If the all files and subdirectories in the former are moved to the latter, the RAD values must be 3. The clone-classes of 6 and 5 are classified as access methods. We investigated clone-classes of 4 and found a clone-pair created in cut-and-paste style, within `javax/ swing/ event/ SwingPropertyChangeSupport.java` and `java/ beans/ PropertyChangeSupport.java`. A class `SwingPrpertyChangeSupport` is directly derived from a parent class `PropertyChangeSupport`, and it contains methods to override those of the parent, but each overridden method is equivalent to the original. The reason for cloning is performance enhancement (the detail is described in the comment of `SwingPropertyChangeSupport`). Therefore, a careful modification process would be required for each of them.

4.1 Application of CCFinder to Linux and FreeBSD systems

CCFinder was applied to million lines of code from two operating systems, Linux 2.2.14 and FreeBSD 3.4. The purpose of this experiment was to investigate where and how similar codes are used between two operating systems. Linux and FreeBSD are well known Unix systems and have independent kernels written in C. The target is the source files of kernel and device-drivers, 2095 `.c` files of 1.6 million lines in Linux, and 2906 `.c` files of 1.3 million lines in FreeBSD. Clone-pairs with 20 LOC or more between two systems are extracted. This operation takes about 40 minutes on the PC.

By investigation of source codes corresponding to the clone-classes of top 30 lengths, such clones belong to 5 files or subsystems, shown in Table 2. The 3 subsystems, `awe_wave`, `mpu401`, and `sequencer` contain files with identical names between two OS's; therefore the mapping of the two OS's for the subsystems could be identified by analysis of file names. On the other hand, '`rocket`' files have different names, `rocket.c` and `rp.c`, so that the identification of the mapping is more difficult.

Table 2. Subsystems cloned between operating systems

Subsystem	Linux files	FreeBSD files
zlib	arch/ppc/coffboot/zlib.c, drivers/net/zlib.c	lib/libz/adler32.c, lib/libz/deflate.c, lib/libz/infblock.c, lib/libz/infcodes.c, lib/libz/inffast.c, lib/libz/inflate.c lib/libz/inftrees.c, lib/libz/trees.c, sys/net/zlib.c
rocket	drivers/char/rocket.c	sys/i386/isa/rp.c
awe_wave	drivers/sound/lowlevel/awe_wave.c	sys/gnu/i386/isa/sound/awe_wave.c
mpu401	drivers/sound/mpu401.c	sys/i386/isa/sound/mpu401.c
sequencer	drivers/sound/sequencer.c	sys/i386/isa/sound/sequencer.c

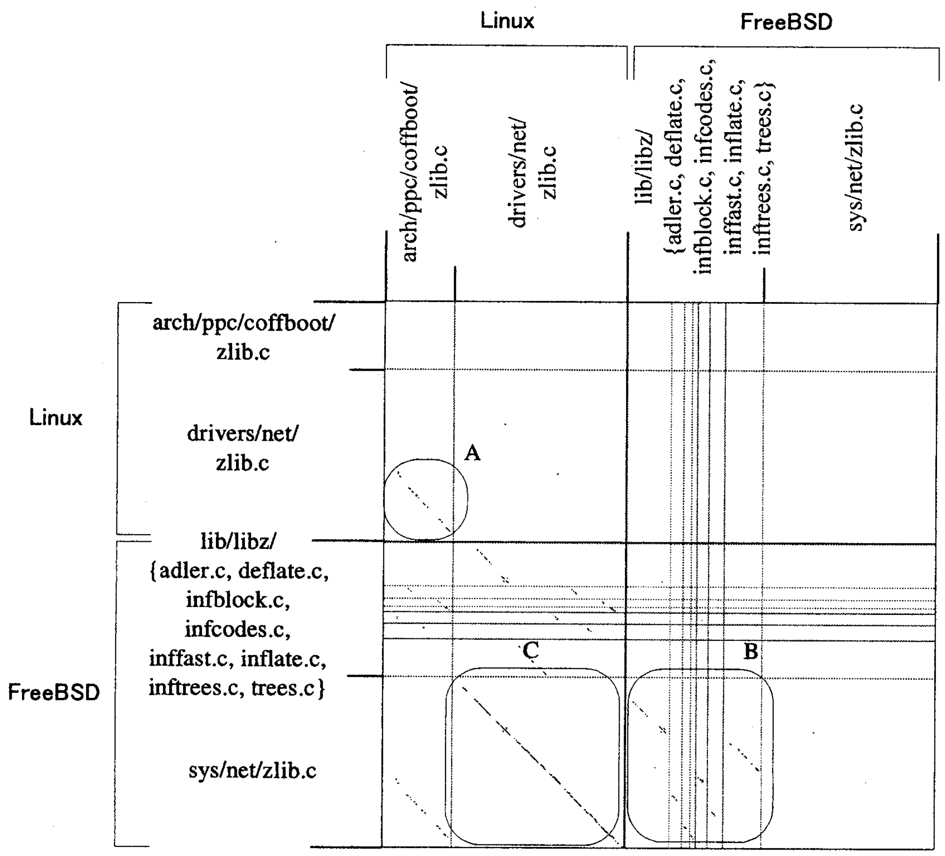

Figure 7 Clones among zlib subsystems.

In case of subsystem `zlib`, the situation is more complex. Linux has two different files with the same name. FreeBSD has 9 files.

Figure 7 shows a scatter plot among the files that have any clones in '`zlib`' files. *A* in the graph shows, Linux has two files named `zlib.c`, and `drivers/net/zlib.c` includes all lines of `arch/ppc/coffboot/zlib.c`. In FreeBSD system, `sys/net/zlib.c` is equal to a concatenation of eight `lib/libz/*.c` files, as shown by *B* in the graph. In both operating systems (OS's), the largest `zlib.c` files contain complete source for '`zlib`' subsystem while the other files contain part of the subsystem. The two largest `zlib.c` files are almost identical between Linux and FreeBSD, as shown by *C*.

5 Conclusions

In this paper, we presented a clone detecting technique with transformation rules and a token-based comparison. We also proposed metrics to select interesting clones. They were applied to several industrial-size software systems in the experiments. An experiment to compare two OS's found several subsystems that would come from a same original. Some of them have distinct file names between OS's, and some are duplicated with in a system.

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] M. G. Bracha et. al. "GJ Specification". http://cm.bell-labs.com/cm/cs/who/wadler/pizza/gj/

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

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

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

[7] 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.

[8] 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.