CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code

Toshihiro Kamiya, *Member*, *IEEE*, Shinji Kusumoto, *Member*, *IEEE*, and Katsuro Inoue, *Member*, *IEEE*

Abstract—A code clone is a code portion in source files that is identical or similar to another. Since code clones are believed to reduce the maintainability of software, several code clone detection techniques and tools have been proposed. This paper proposes a new clone detection technique, which consists of the transformation of input source text and a token-by-token comparison. For its implementation with several useful optimization techniques, we have developed a tool, named CCFinder, which extracts code clones in C, C++, Java, COBOL, and other source files. As well, metrics for the code clones have been developed. In order to evaluate the usefulness of CCFinder and metrics, we conducted several case studies where we applied the new tool to the source code of JDK, FreeBSD, NetBSD, Linux, and many other systems. As a result, CCFinder has effectively found clones and the metrics have been able to effectively identify the characteristics of the systems. In addition, we have compared the proposed technique with other clone detection techniques.

Index Terms—Code clone, duplicated code, CASE tool, metrics, maintenance.

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 "copy-and-paste," mental macro (definitional computations frequently coded by a programmer in a regular style, such as payroll tax, queue insertion, data structure access, etc.), or intentionally repeating a code portion for performance enhancement, etc. [5]. A conservative and protective approach for modification and enhancement of a legacy system would introduce clones. Also, systematic generation of a set of slightly different code portions from a single basis will bear clones. Clones make the source files very hard to modify consistently. For example, let's 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 [15]. For a large and complex system, there are many engineers who take care of each subsystem and then modification becomes very difficult. If the existence of clones has been documented and maintained properly, the modification would be relatively easy; however, keeping all clone information is generally a laborious and expensive process. Various clone detection tools have been proposed and implemented [1], [2], [3], [4], [5], [7], [11], [14], [15], [17] 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 [17].

We were interested in applying a clone detection technique to a huge software system for a division of government, which consists of one million lines of code in 2,000 modules written in both COBOL and PL/I-like language, which was developed more than 20 years ago and has been maintained continually by a large number of engineers [18], [21]. It was believed that there would be many clones in the system, but the documentation did not provide enough information regarding the clones. It was considered that these clones heavily reduce maintainability of the system; thus, an effective clone detection tool has been expected.

Based on such initial motivation for clone detection, we have devised a clone detection algorithm and implemented a tool named CCFinder (Code clone finder). The underlying concepts for designing the tool were as follows:

- The tool should be industrial strength and be applicable to a million-line size system within affordable computation time and memory usage.
- A clone detection system should have the ability to select clones or to report only helpful information for user to examine clones since large number of clones is expected to be found in large software systems. In other words, the code portions, such as short ones inside single lines and sequence of numbers for table initialization, may be clones, but they would not be useful for the users. A clone detection system that removes such clones with heuristic knowledge improves effectiveness of clone analysis process.

- *T. Kamiya is with Functions and Configuration Group, RPESTO, JST. Graduate School of Engineering Science, Osaka University, 1-3 Machikaneyama-cho, Toyonaka, Osaka 560-8531, Japan. E-mail: kamiya@ics.es.osaka-u.ac.jp.*
- *S. Kusumoto and K. Inoue are with the Graduate School of Engineering Science, Osaka University, 1-3 Machikaneyama-cho, Toyonaka, Osaka 560-8531, Japan. E-mail: {kusumoto, inoue}@ics.es.osaka-u.ac.jp.*

Manuscript received 19 July 2000; revised 28 Mar. 2001; accepted 17 Sept. 2001.
Recommended for acceptance by L. Briand.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 112550.

0098-5589/02/$17.00 © 2002 IEEE

- Renaming variables or editing pasted code after copy-and-paste makes a slightly different pair of code portions. These code portions have to be effectively detected.
- The language dependent parts of the tool should be limited to a small size and the tool has to be easily adaptable to many other languages.

The tool makes a token sequence form the input code through a lexical analyzer and applies the rule-based transformation to the sequence. The purpose is to transform code portions in a regular form to detect clone code portions that have different syntax but have similar meaning. Another purpose is to filter out code portions with specified structure patterns. Representing a source code as a token sequence enables us to detect clones with different line structures, which cannot be detected by line-by-line algorithm.

The language-dependent parts of the tool are designed as small replaceable components. The first version of the tool had components for C and C++. Transformation rules and components for Java were developed in two person days and those for COBOL were developed in one week.

We have used a *suffix-tree matching algorithm* [10] to compute matching, in which the clone location information is represented as a tree with sharing nodes for leading identical subsequences and the clone detection is performed by searching the leading nodes on the tree. Our token-by-token matching is more expensive than line-by-line matching in terms of computing complexity since a single line is usually composed of several tokens. However, we propose several optimization techniques especially designed for the token-by-token matching algorithm, which enable the algorithm to be practically useful for large software.

The clone metrics presented in this paper refer to an equivalence class of clones (defined in Section 2.1) and measure where and how frequently clones appear. The metrics enable us to estimate how many lines are reduced by removing the clone codes and to evaluate how far the clone-containing files are spread over the hierarchy file system.

The applications of our tool are also a novel contribution of this paper. We have applied the CCFinder to various kinds of source codes in various languages, such as Unix system in C, JDK in Java, and the governmental system in COBOL and a PL/I-like language. The effectiveness of the approach has been evaluated quantitatively and qualitatively. The similarity between Linux, FreeBSD, and NetBSD, as well as the nature of JDK, has also been explored. CCFinder and the metrics have detected clones that are small in size by themselves but many lines in them would be removed by rewriting them using a shared routine.

The major contributions of our approach are as follows:

- Our token-based clone detection method, including various optimization techniques, is very efficient and scalable to large software systems. The largest case study we have tried is the source code of two versions of OpenOffice [20], 10 million lines in total. The trial took 68 minutes (elapsed time) on a computer with 650MHz Pentium III with 1 GBytes memory.
- Our approach is very easily adaptable to many other programming languages. The lexical analyzer and the transformation rules are the only language-dependent parts in the tool and they ware easily constructed in a few days. At this moment, our tool CCFinder accepts, C, C++, Java, and COBOL.
- Our approach detects clones that cannot be detected by other methods. Clones with different line breaks or different identifier names are detected. Our case studies presented in Section 4.2 shows that more than 23 percent of the clones are detected compared to the line-by-line method proposed in [1].
- We have applied CCFinder to various source codes to find clones inside single systems and to explore differences or similarity of two or more systems.

In this paper, Section 2 defines clones and explains our clone-detecting algorithm and metrics for clones. In Section 3, the clone detection method and the metrics are evaluated empirically with several software systems of industrial size. Section 4 surveys related works and discusses our approach. Finally, Section 5 concludes the paper and briefly presents future work.

2 Proposed Clone-Code Detection Technique

2.1 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 clonerelation 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 subportions of code portions in each clone class also make clone classes (e.g., Each of C3 is a subportion of C1). In this paper, however, we are interested only in the maximal portions of clone classes so we only discuss the maximal portions (e.g., C1). The three code portions in C2 constitute a clone class since the third one is not a subsequence of any code portions of C1.

In studies [5], [7], [11], [12], [15], [17], the clone analyses have been based on clone pairs. On the other hand, the clone analysis in our approach, as well as in [1] and [3], use both clone pairs and clone classes. The analysis by clone classes and the metrics are described in Section 2.4.

In our approach, identifying the clone relation is made for the transformed token sequences in order to extract many clones whose token sequences are slightly modified. In the following sections, a "clone" means a code portions having another code portion that might be different in

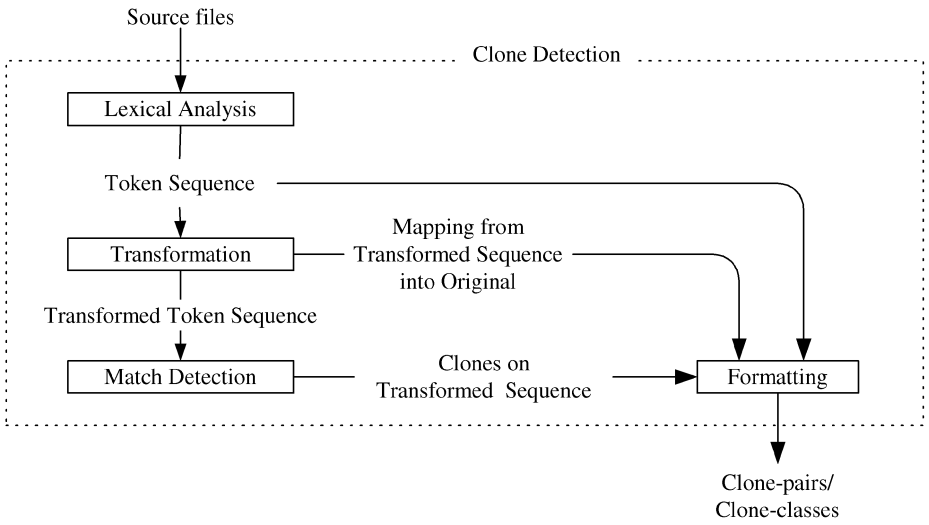


Fig. 1. Clone detecting process.

verbatim way but is the same as transformed token sequences.

2.2 Some Issues in Clone Detection

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

2.2.1 Identification of Structures

It should be noticed that certain types of clones seem difficult to rewrite as a shared code even if they are found as same code portions. 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 and, in [17], only an entire function definition can become a candidate for clone.

2.2.2 Regularization of Identifiers

Recent programming languages such as C++ and Java provide *name space* and/or *generic type* [6]. 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 front-end of the clone detection tool, the lexical analysis and transformation, should include all language-dependent parts; whereas, the back-end, the matching detection, should be language-independent. The details of the clone detecting process are described in Section 2.3.

2.2.3 Measuring Clones

Large software systems often include many clones, so a clone analysis method must distinguish important clones from many "uninteresting" clones. Baker estimated how many lines of code are reduced when clone code portions of C source files are rewritten by macro [1]. Baxter et al. investigated the percentage of clone code in a software system and proposed using clone detection to capture domain idiom in a system [5]. Balazinska et al. proposed 18 categories based on differences between code portions of a clone pair. A qualitative classification of clones to get an index for reengineering opportunity, which would require further elaboration for tool support, was proposed in [3].

In Section 2.4, we present metrics for clone classes to evaluate how wide code portions of a clone class are distributed in a system or to estimate the reduction of code by a clone class. The metrics enable one to identify important clones, such as clones that enable large code reduction by their removal or clones that have so widely spread in the system that it is difficult to find and maintain them by hand.

2.3 Clone-Detecting Process

Clone detection 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 Fig. 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 (including \n and \t and comments) between tokens are removed from the token sequence, but those characters are sent to the formatting step to reconstruct the original source files.

TABLE 1
Transformation Rules for C++

Rule #	Rule description
RC1 Remove namespace attribution	**(Name '::')+ Name2 → Name2** Here, the operator +, a postfix operator of regular expression, means repeat of one or more times. In C++ source files, a name may belong to a name space or a class and can be spelled in full or in shorter form. The transformation is to neglect the attribution so that they are considered equivalent in clone detection. Ex. `std::ios_base::hex` is transformed into `hex`.
RC2 Remove template parameters	**Name '<' ParameterList '>' → Name** Here, ParameterList is a sequence of Name, Number, String, Operators, ',' and Expression. Expression is a sequence of tokens which starts with '(' and ends with the corresponding ')' and does not include ';'. Template arguments may be omitted because of a type estimation by the compiler or because of the scope of the template. The transformation copes with the case. Ex. `sort<int>` is transformed into `sort`.
RC3 Remove initializtion lists	**'=' '{' InitalizationList, '}' → '=' '{' UniqueIdentifier '}'** Here, InitalizationList is a sequence of Name, Number, String, Operators, ',', '(', ')', '{', and '}'. UniqueIdentifier is a unique token, which never appears in another place of a token sequence. Some tables (such as character code, color code, and wave table) include a continuation of a value and regular repeats of some values. The rule eliminates such large table initialization codes.
RC4 Separate function definitions	**Insert UniqueIdentifier at each end of the top-level definitions and declaration.** This rule prevents extraction of clone pairs of the code portions that begin at the middle of a function definition and end at the middle of another function definition. Similar ideas are found in [4] and [13].
RC5 Remove accessibility keywords	**AccessibilityKeyword → *Φ*** Here, phi is a null sequence. Ex. `protected: void foo();` is translated to `void foo();`.
RC6 Convert to compound block	**Each single statement after if (), do, else, for (), and while () is transformed to a compound block.** Ex. `if (a == 1) b = 2;` is transformed to `if (a == 1) { b == 2; }`

2. **Transformation**. The token sequence is transformed with 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 formatting step which comes later.

   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. Tables 1 and 2 show the transformation rules aiming at regularization of identifiers (RC1, RC2, RJ1, RJ2, and RJ5) and identification of structures (RC3, RC4, RJ3, and RJ4). Rules such as RJ1 and RC1 partially remove context information of the source program, although those are very effective and essential as discussed in Section 3.3.

   2.2. **Parameter Replacement**. After Step 2.1, each identifier related to types, variables, and constants is replaced with a special token. This replacement makes code portions with different variable names to become clone pairs.

3. **Match 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 (LeftBegin, LeftEnd, RightBegin, RightEnd), where LeftBegin and LeftEnd are the beginning and termination positions (indices in the token sequence) of a leading clone, and RightBegin and RightEnd belong to another following clone for a clone pair.

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

Fig. 2 shows an example input of a C++ source to explain the clone-detecting process. The numbers at the left of the figure are line numbers. The input is divided into tokens. The token sequence transformed by the transformation rules is shown in Fig. 3. Lines 1, 3, 11, and 13 become shorter. After this step, the token sequence is transformed again by parameter replacement. The token sequence after parameter replacement is shown in Fig. 4. Identifiers are replaced with a token $\$p$ in the sample. Finally, clone pairs, i.e., equivalent substrings in the token sequence, are identified. Let t_i denote the ith token $(1 \leq i \leq 114)$ in the token sequence in Fig. 4 and let us make a matrix $\{ d_{xy} \}$, where $d_{xy} = 1$ if t_x is equal to t_y, 0 otherwise. The part of the matrix is shown in Fig. 5. In this figure, we place "*" for $d_{xy} = 1$ when $x > y$. Since it always holds that $d_{xy} = d_{yx}$ (symmetric) and $d_{xx} = 1$ for any x and y, we place nothing for $d_{xy} = 1$ when $x \leq y$. A clone pair is found as a line

TABLE 2
Transformation Rules for Java

Rule #	Rule description
RJ1 Remove package names	**(PackageName '.')+ ClassName → ClassName** Here, PackageName is a word that begins with a small letter and ClassName is a capitalized word. In Java source files, a class is referred to with either the full package name or a shorter name by using import sentences. The transformation is to neglect the attribution so that they are considered equivalent in clone detection. Ex. `java.lang.Math.PI` is transformed to `Math.PI`.
RJ2 Supplement callees	**NDotOrNew NClassName '(' → NDotOrNew CalleeIdentifier '.' NClassName '('** Here, NDotOrNew is a token except '.' or 'new'. NClassName is an uncapitalized word. CalleeIdentifier is a token for an omitted callee. By language specification a method is either an instance method or a class method. Therefore, if an instance calls a method without a callee instance or class then the omitted callee is the instance itself or a class of it.
RJ3 Remove initializtion lists	**'=' '{' InitalizationList, '}' → '=' '{' '}'** **']' '{' InitalizationList, '}' → ']' '{' '}'** Here, InitalizationList is a sequence of Name, Number, String, Operators, ',', '(', ')', '{', and '}'. These rules are an expansion of rule RC3. The second rule is applied where an array is created with initialization by a new expression. For example, `return new int[] { 1, 2, 3 };`.
RJ4 Separate class definitions	**Insert UniqueIdentifier at each end of the top-level definitions and declaration.** This rule prevents extracting clone pairs of the code portions that begin at the middle of a class definition and end at the middle of another class definition.
RJ5 Remove accessibility keywords	**AccessibilityKeyword → *Φ*** Here, phi is a null sequence. Ex. `protected void foo()` is translated to `void foo()`.
RJ6 Convert to compound block	**Each single statement after if (), do, else, for (), and while () is transformed to a compound block.** Ex. `if (a == 1) b = 2;` is transformed to `if (a == 1) { b = 2; }`

segment of "*" that is parallel to the main diagonal of the matrix. The code portions from line 1 to 7 and from line 11 to 17 make a clone pair. The code portions from line 8 to 10 and from 19 to 21 make another clone pair. Subportions of those clones can also be clone pairs; however, we are only interested in those of the maximum length and the tool does not report their subportions.

```
 1| Void print_lines(const set<string>& s) {
 2|     int c = 0;
 3|     set<string>::const_iterator i
 4|         = s.begin();
 5|     for (; i != s.end(); ++i) {
 6|         cout << c << ", "
 7|             << *i << endl;
 8|         ++c;
 9|     }
10| }
11| Void print_table(const map<string, string>& m) {
12|     int c = 0;
13|     map<string, string>::const_iterator i
14|         = m.begin();
15|     for (; i != m.end(); ++i) {
16|         cout << c << ", "
17|             << i->first << " "
18|             << i->second << endl;
19|         ++c;
20|     }
21| }
```

Fig. 2. Sample code.

Here, a clone-relation is defined with the transformation rules and the parameter-replacement described above. Other clone relations can be defined with different transformation rules or by neglecting the parameter replacement. In the case studies described in Section 3, a

```
 1| void print_lines ( const set & s ) {
 2| int c = 0 ;
 3| Const_iterator I
 4| = s . begin ( ) ;
 5| for ( ; i != s . end ( ) ; ++ i ) {
 6| cout << c << ", "
 7| << * I << endl ;
 8| ++ c ;
 9| }
10| }
11| void print_table ( const map & m ) {
12| int c = 0 ;
13| Const_iterator I
14| = m . begin ( ) ;
15| for ( ; i != m . end ( ) ; ++ i ) {
16| cout << c << ", "
17| << i -> first << " "
18| << i -> second << endl ;
19| ++ c ;
20| }
21| }
```

Fig. 3. The transformed code by the transformation rules.

```
 1| $p $p ($p $p & $p ) {
 2| $p $p = $p ;
 3| $p $p
 4| = $p . $p ( ) ;
 5| for ( ; $p != $p . $p ( ) ; ++ $p ) {
 6| $p << $p << $p
 7| << * $p << $p ;
 8| ++ $p ;
 9| }
10| }
11| $p $p ($p $p & $p ) {
12| $p $p = $p ;
13| $p $p
14| = $p . $p ( ) ;
15| for ( ; $p != $p . $p ( ) ; ++ $p ) {
16| $p << $p << $p
17| << $p -> $p << $p
18| << $p -> $p << $p ;
19| ++ $p ;
20| }
21| }
```

Fig. 4. The code after parameter replacement.

clone relation with all the transformation rules is compared to a clone relation with a subset of the transformation rules.

2.4 Metrics for Evaluating Clone Pairs and Clone Classes

For quantitative evaluation, selection, and filtering of clones, we define several metrics for clone classes. These metrics enable us to answer questions, such as "which clone class will bring the largest reduction of code by rewriting with a shared routine?" or "which clone code portion is widely or frequently used in the system?"

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

LEN(p) is the length of a code portion of p. *LEN*(C) for clone class C is the maximum *LEN*(p) for each p in C. The length can be measured by the number of tokens, or the size measures such as LOC (the number of lines, including null and comment lines), or SLOC (the number of lines, except null or comment lines). In our approach, the number of tokens of each code portion of a clone class is identical when it is measured on a transformed token sequence;

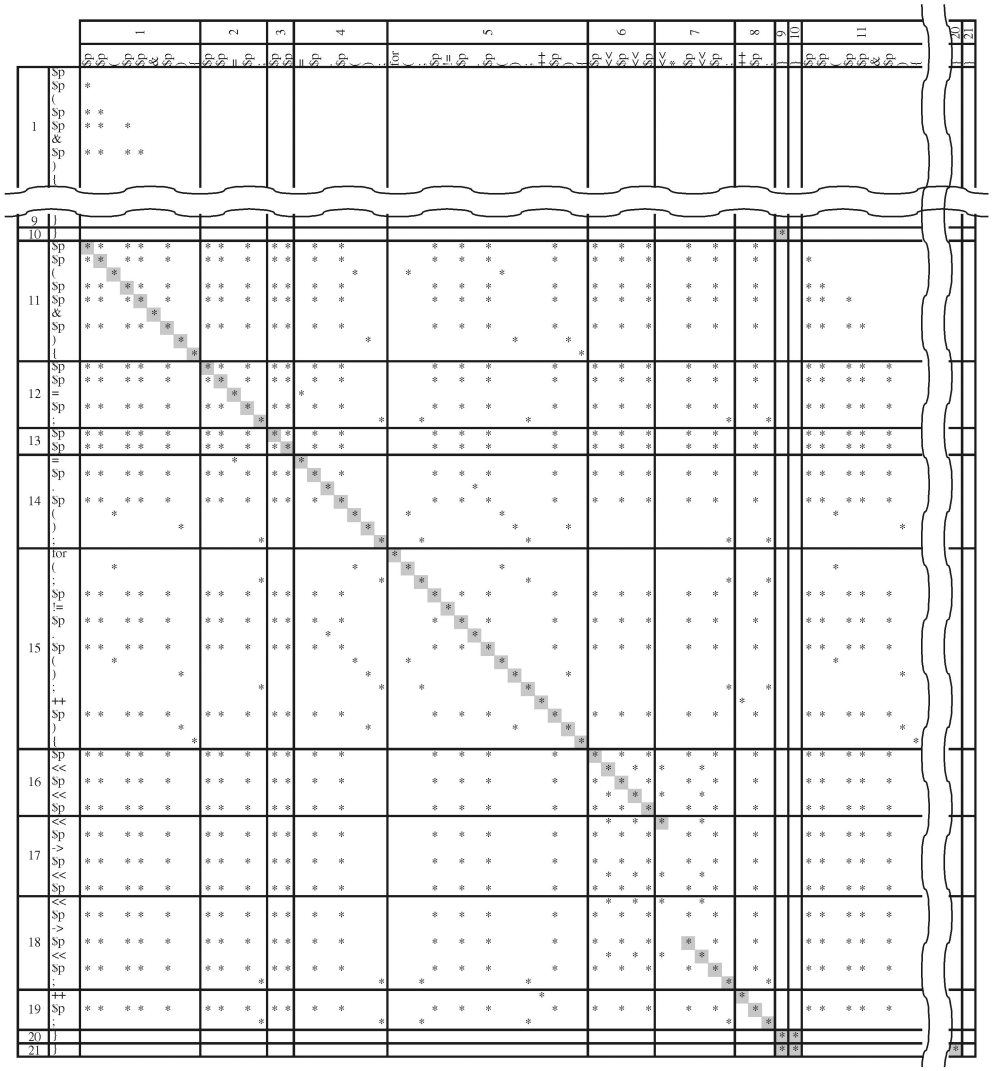
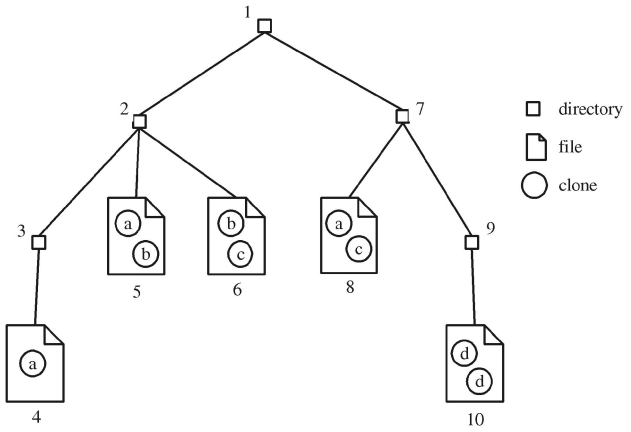


Fig. 5. Matrix showing the scatter plot token-by-token.



Clone class C	$RAD(C)$
{ 4:a, 5:a, 8:a }	3
{ 5:b, 6:b }	1
{ 6:c, 8:c }	2
{ 10:d, 10:d }	0

Fig. 6. Distance, radius and population.

however, they may have different numbers when measures as the original tokens, LOC or SLOC. We use LOC as the unit of LEN in the following since we can easily capture the appearance size of clones in the source code.

2.4.2 Population of a 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.

2.4.3 Deflation by a Clone Class; DFL(C)

Combination of LEN and POP gives an estimation of the amount of code which 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 shared routine (function, method, template function, or so) and that the length of this caller statement is $USELEN(C)$. In this case, $LEN(C) \times POP(C)$ code portions are occupied in the original source files. In the newly restructured source files, $USELEN(C) \times POP(C)$ is required for caller statements and $LEN(C)$ for a callee routine. Now, let us define a metric DFL as an estimator of reduced source lines:

$$\begin{aligned} DFL(C) &= (\text{old } LOC \text{ related to } C) - (\text{new } LOC \text{ related to } C) \\ &= LEN(C) \times POP(C) \\ &\quad - (USELEN(C) \times POP(C) + LEN(C)). \end{aligned}$$

2.4.4 Coverage of Clone Code; COVERAGE (% LOC) and COVERAGE (% FILE)

These metrics are defined for all clones in a whole source code, not for one particular clone pair or clone class. COVERAGE (% LOC) is the percentage of lines that include any portion of clone, and COVERAGE (% FILE) is the percentage of files that include any clones.

2.4.5 Radius; RAD(p), RAD(C)

For a given clone class C, let F be 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. If all code portions of C are included in one file, $RAD(C) = 0$. In Fig. 6, RAD({4:a, 5:a, 8:a }) = 3 since their lowest common ancestor is 1 and the maximum path length from directory 1 to each file is 3 for file 4. Note that RAD({ 10:d, 10:d }) = 0 since the lowest common ancestor is file 10 itself.

Each metric measures different aspects of clones. DFL measures an impact in size of reduction of each clone class, while RAD measures extent of influence of a clone class. If a clone class has a large RAD, the code portions widely spread over a software system and it would become difficult to maintain their consistency correctly since such different subsystems are likely to be maintained by different engineers. If we use clone detection to find idioms (or mental macros) [5], a clone with small RAD value is a "local" idiom, whose code portions are peculiar to a limited area of a system, while a clone with large RAD value is a "global" idiom of the system.

COVERAGE is used to evaluate similarity between two software (sub)systems, and to compare two versions of a (sub)system [5], [15].

2.5 Implementation Techniques of Tool CCFinder

Tool CCFinderhas been implemented in C++ and runs under Windows 95/NT 4.0 or later. CCFinder extracts clone classes from C, C++, Java, and COBOL source files. The tool receives the paths of source files and writes the locations of the extracted clone classes to the standard output. CCFinderuses a suffix-tree algorithm with both time and space complexities O(m n), where m is the maximum length of involved clones and n is the total length of the source file. If we would naturally assume that m does not depend on n and it is bounded by some fixed length, the time and space complexities will practically be O(n). Fig. 7 and Fig. 8 show actual measurement of time and space complexiteis, when the tool is executed on PC with Pentium 4 1.5GHz and 640MB RAM, given various sized subsets of source files of Linux 2.4.9 up to 2,600k lines.

The optimizations employed inCCFinder to handle large source files are as follows:

- **Alignment of Token Sequence**. Source code has its inherent granularity such as character, token, statement, or block. Code portions of a code clone should begin at their boundary. For example, a code

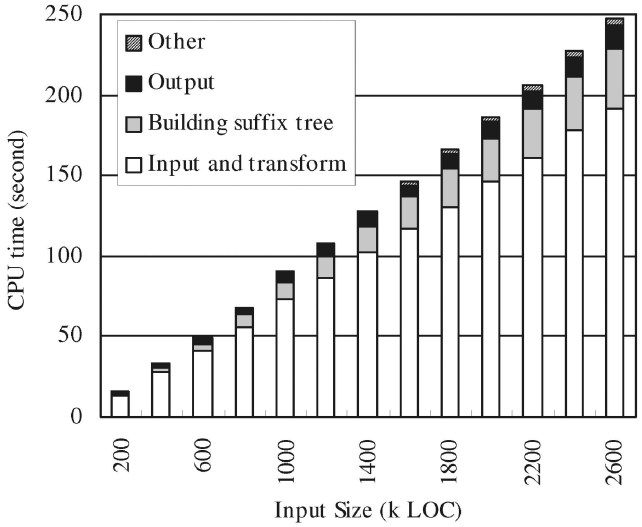


Fig. 7. CPU time of CCFinder.

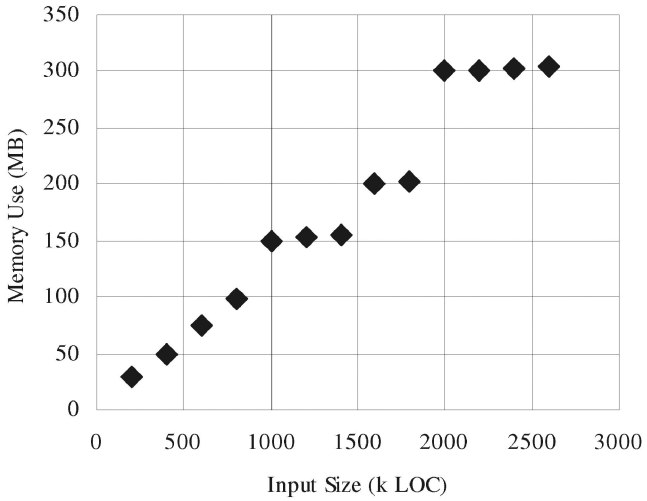


Fig. 8. Memory usage of CCFinder.

portion, which begins at the middle of a statement X and ends the middle of a statement Y, is less useful than a code portion which begins at the beginning of Y. As a simple filtering for this purpose, we allow only specific tokens at the beginning of clones as leading tokens. Keywords that initiate statements are leading tokens. In C and C++ source files, those keywords are "#", "{", keywords for selection statements (else, if, switch, etc.), iteration statements (do, for, and while), jump or structured exception handling statements (break, catch, return, etc), and declarations (class, enum, typedef, etc). Also, tokens following keywords that terminate statements (";", ")"} or labels (":") are also leading tokens. The number of nodes in the suffix tree was reduced to one third by this filtering in C, C++, and Java case studies described in Section 3. This technique might slightly reduce the sensitivity of clone detection, but practically it is very important to make the tool scalable.

- **Repeated Code Removal**. Repetition of a short code portion tends to generate many clone pairs. For example, consider the following code:

```
a1 |    switch (c) {
a2 |    case '0' : value = 0; break;
a3 |    case '1' : value = 1; break;
a4 |    case '2' : value = 2; break;
a5 |    case '3' : value = 3; break;
a6 |    case '4' : value = 4; break;
a7 |    }
```

Now, consider that the following code section is also included in the target source files:

```
b1 | case 'a':
b2 |    flag = 2;
b3 |    break;
```

In this case, five code portions make a clone class, { a2-a2,a3-a3, ..., a6-a6, b1-b3 }, where each pair of the code portions makes a clone pair, and the number of maximal clone pairs are ${}_6C_2 = 15$, in total. To avoid this explosion of clone pairs, a heuristic approach is introduced. Upon building a suffix-tree, if a repetition of a2 is identified at a3, the succeeding repetition section (a3-a6) is not intentionally inserted into the tree, so that a part of the clone pairs is not being reported. However, the clone pair (a2-a2, b1-b3) is still extracted, which offers sufficient information. The repeated code removal process also prevents detection of self clones, e.g., (a2-a5, a3-a6), or repetition of "constant"declarations.

- **Concatenation of Tokens**. Just before computing the match in the token sequence, abutting tokens, except for punctuator keywords, are concatenated. This process reduces the length of a token sequence in exchange for an increase in variation of the tokens.
- **Division of Large Archive of Source Files**. If the total size of source files exceeds the memory space for a single suffix-tree, the tool automatically employs a "divide and conquer" approach. The input source files are divided into several parts. For each combination of the parts, a subsuffix tree is built to extract clone pairs. The total collection of clone pairs will finally be the output. Let m be the number of subsets of source files, and then the number of pairs of the chunks (i.e., the number of constructed subsuffixtrees) is ${}_mC_2$. Therefore, the time complexity becomes $O(m^2)$. However, our system allows about 2.6 million C source lines without division under 640 MByte memory and five million lines without division under 1GByte memory. Even if division is needed, the tool can handle each part very fast and the total execution time is practically acceptable. For example, we have analyzed 10 million lines from two versions of OpenOffice [20]. To detect clones, the input was divided into eight chunks with 640 MByte memory and it took two hours. Also, it was divided with 1 GByte memory into four chunks and it took 68 minutes.

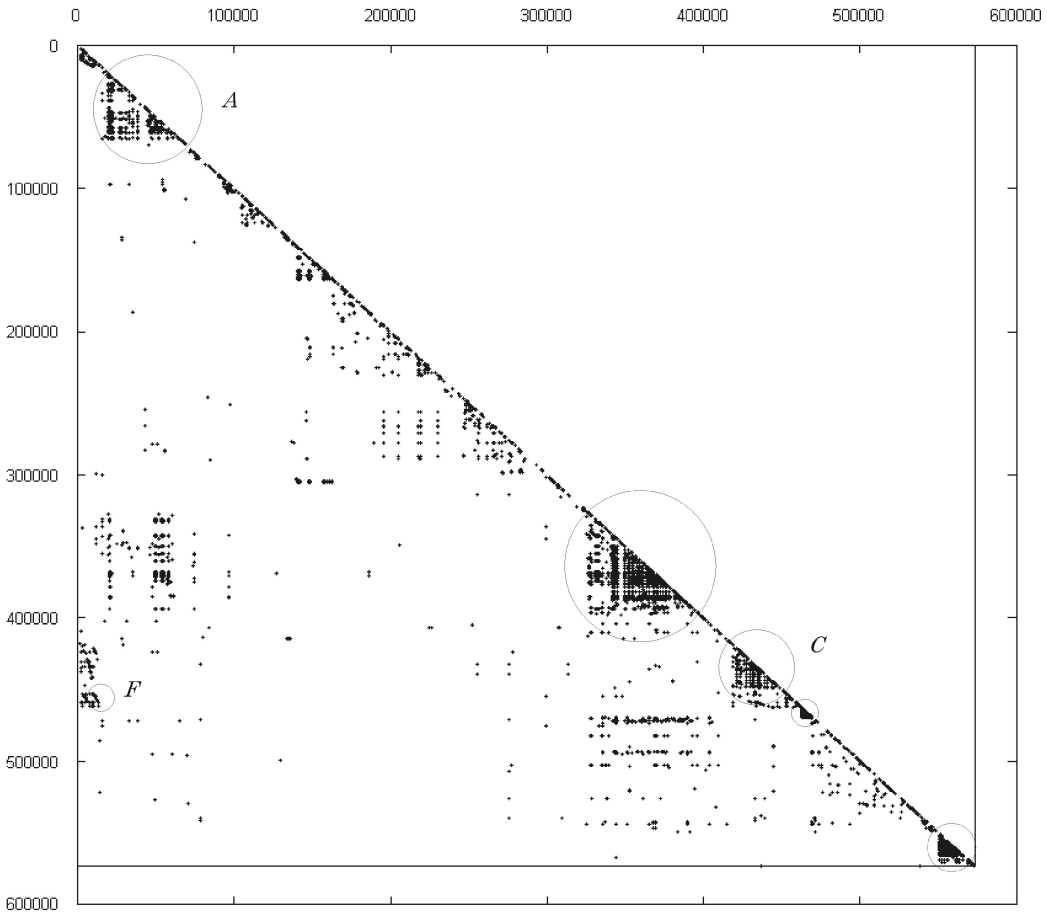


Fig. 9. Scatter plot of clones over 30 transformed tokens in JDK 1.3.0.

3 Case Studies

The purpose of the case studies was to evaluate our token-based clone-detecting technique and the metrics. The target source files were widely available files of "industrial" size. In all the case studies, CCFinder was executed on a PC with Pentium III 650MHz and 640MB RAM, which seem to be moderate, nonspecial hardware specification for PC these days. In the following discussion, we will use elapsed time on this PC.

3.1 Clones in a Java Library, JDK

3.1.1 Overview

JDK 1.3.0 [22] is a commonly used Java library and the source files are publicly available. Tool CCFinder has been applied to all source files of JDK, about 570k lines in total, in 1,877 files. It takes about three minutes for execution on the PC. Fig. 9 shows a scatter plot of the clone pairs having at least 30 same tokens (about 13 lines). Both the vertical and horizontal axes represent lines of source files. The files are sorted in alphabetical order of the file paths, so that files in the same directory are also located nearby 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 2.3. In Fig. 9, each line segment looks like a dot since each clone pair is small (average 39, up to 628 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.

There are several crowded areas, marked *A*, *B*, *C*, *D*, and *E*. Area *A* corresponds to source files of `java/awt/*.java`, *B*, *C, and D* to `javax/swing/*.java`, and *E* to `org/omg/ CORBA/*.java`. *D* contains many "clone files," that is, very similar source files. Some of them contain an identical class definition except for their different parent classes. Fig. 10 shows parts of the two files as examples, namely, `MultiButtonUI.java` and `MultiColorChooserUI.java`. Differences are only in lines 32, 161, and 163. According to the comments of the source files, a code generator named `AutoMulti` has created these files. To modify them, the developer should obtain an automatic code generation tool called `AutoMulti` (it is not included in JDK), edit, and apply it correctly. If the developer does not have the tool, all the files have to be updated carefully by hand. In this case, these code portions have two different names—the base classes and the type of local variables named `mui`. Redesign techniques for Java are presented in [4] and might be applicable to this case. Also, using *generic type* for Java, as proposed in [6], would enable to rewrite them as a shared code.

The longest clone (1,647 tokens, 627 lines) was found between `src/com/sun/java/swing/plaf/windows/WindowsFileChooserUI.java` and `src/javax/swing/ plaf/metal/MetalFileChooserUI.java` (marked *F* in Fig. 9). Each of the two classes `WindowFileChooserUI` and `MetalFileChoosrUI` has nine

```
 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|     }
```

(a)

```
 31| */
 32| public class MultiColorChooserUI extends ColorChooserUI {
 33|

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

(b)

Fig. 10. A pair of similar source files found in JDK. (a) MultiButtonUI.java. (b) MultiColorChooserUI.java. These two files are identical except for three identifiers shown in bold style.

internal classes, one constructor, and 45 methods, all of which, except three of the methods, are clones.

3.1.2 Evaluation of Transformation Rules for JDK

In Section 2.3, we also proposed the transformation rules for Java. To evaluate the effectiveness of the transformation rules (i.e., how many clones useful for redesigning the system are found), we have applied CCFinder with some of their transformation rules disabled. Table 3 shows the amount of clone pairs and clone classes, COVERAGE (% LOC) and COVERAGE (% FILE), when all or part of rules are applied. Case $P+R+123456$, the default options of CCFinder, means that the parameter replacement(P), repeated code removal(R) and all rules (from RJ1 to RJ6) are applied. Case $P+12456$ lacks processes, Repeated Code Removal and RJ3. A special filtering to remove self clones is introduced for this case. (It should be kept in mind that self clones are usually removed by the process of repeated code removal). Case R intends to apply only repeated code removal, with no parameter-replacement, no Repeated Code Removal, or no transformation (but white spaces, line breaks, and comments are still removed). In this case study, the clone pairs found by $P+R+123456$ are much fewer than with $P+12456$. This means that rule Repeated Code Removal and RJ3 remove many table initialization codes.

There is little difference in amount of clone classes or clone pairs between $P+R+123456$ and $P+R+34$, although COVERAGE percentages slightly differ. Fig. 11 shows one of code portions detected with transformation rule RJ2. The lower code portion has a method call with a class name (`Utility.arranRegiionMatches`), while the upper code portion has a call without class name (`arrayRegionMatches`). When no transformation rules, no parameter replacement, or no repeated code removal are applied (case R), about half of the clone pairs are not detected.

3.1.3 Analysis Using Clone Metrics

The clone classes of JDK, 2,333 in total, were analyzed using the metrics. Fig. 12 shows the LEN and POP parameters of each clone class. The set of clone classes with the highest 5 percent DFL values are 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 5 percent DFL values are classified into the following four types:

- Sequence of methods (45 clone classes).
- A method body or a repeat of clone method bodies (28 clone classes).

TABLE 3
Effects of Transformation Rules and Other Preprocessing Techniques for JDK

	P+R+123456	*P+12456*	*P+R+34*	*R*
# clone pairs	8047	51825	8063	3842
# clone classes	2333	4638	2324	1038
COVERAGE (% LOC)	21.35%	29.15%	20.51%	8.76%
COVERAGE (% FILE)	46.24%	53.38%	45.34%	27.49%

"P+R+123456' means that parameter replacement, repeated code removal, and the transformation rule RJ1 to RJ6 are applied.

```
if (hashes[i] == hashes[j] &&
       arrayRegionMatches(values, iBlockStart,
       values, jBlockStart, BLOCKCOUNT)) {
    indices[i] = (short)jBlockStart;
    break;
}
```

```
if (hashes[i] == hashes[j] &&
       Utility.arrayRegionMatches(values, iBlockStart,
       values, jBlockStart, BLOCKCOUNT)) {
    indices[i] = (short)jBlockStart;
    break;
}
```

Fig. 11. Part of a clone pair captured by rule RJ2.

- Source files generated by a code generation tool `AutoMulti` (22 clone classes).
- Routines within a method (13 clone classes).
- An entire class body (9 clone classes).

This result reveals that there are a number of granularities of reusing code in copy-and-paste style in JDK, namely, routines in a method, single method bodies, sequences of several method bodies, and entire class bodies.

Fig. 13 shows the RAD and POP values of each clone class with at least 20 tokens (about 8 lines). *A* includes "local" clones of source files generated by `AutoMulti`. *B* and *C* include exception classes, which are globally found in JDK libraries. All clone classes which have RAD value 7 (marked *D*) are found in the "`swing`" subsystem, which has source files located at distant directories, `src/com/sun/java/swing` and `src/javax/swing`. If the all files and subdirectories in the former would move to the latter, the RAD values were three. This result shows that clones tend to occur 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 since the developer needs to search for the target code portion through many files. For example, the source file `MultiButton.java` has 1, 29, 162, 506, 616, or 1,877 neighbor files for each RAD value from 0 to 5, respectively. Another reason would be that the nearer files are more likely to implement similar functionalities.

3.2 Similarity to FreeBSD, Linux, and NetBSD Systems

CCFinder was applied to more than one million lines of code of three operating systems, namely, FreeBSD 4.0 [8], Linux 2.4.0 [16], and NetBSD 1.5 [19]. FreeBSD and NetBSD are branches of BSD OS, whereas Linux has another code origin. The case study was intended to investigate where and how similar codes are used among three operating systems. They consist of kernels and drivers written in C. The target is the source files of their kernel and device drivers, in 4,863 files in 2.2 million lines in FreeBSD, 3,240 files in 2.4 million lines in Linux, and 5,928 files in 2.6 million lines in NetBSD. Clone pairs with 30 tokens or more in three systems are extracted. The operations take 108 minutes on the PC, in total.

Fig. 14 shows the scatter plot of clones among the three abovementioned systems (FreeBSD, Linux, and NetBSD).

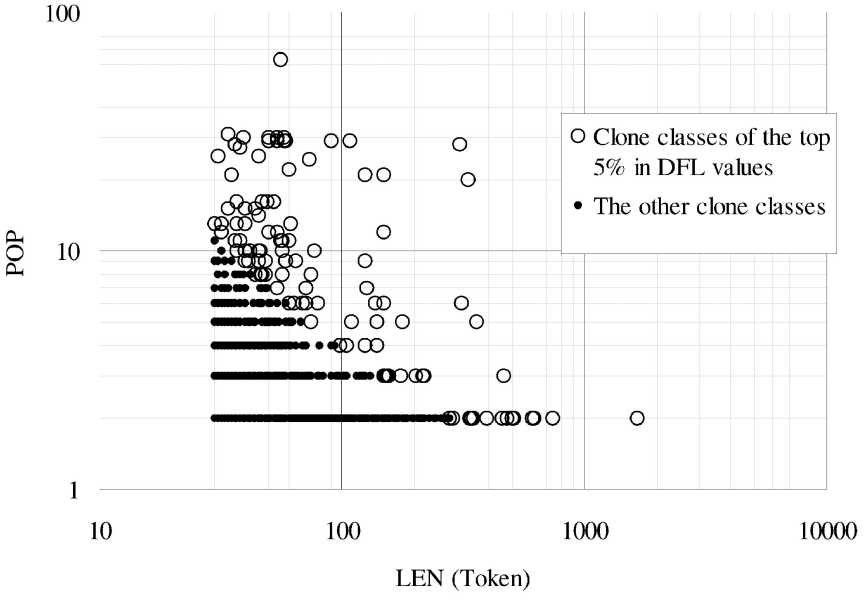


Fig. 12. Population and length of clone classes in JDK.

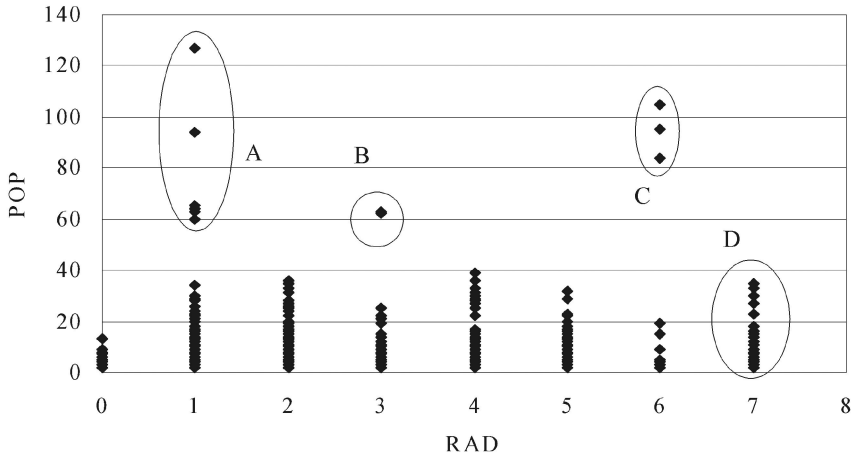


Fig. 13. Population and radius of clone classes over 20 transformed tokens in JDK.

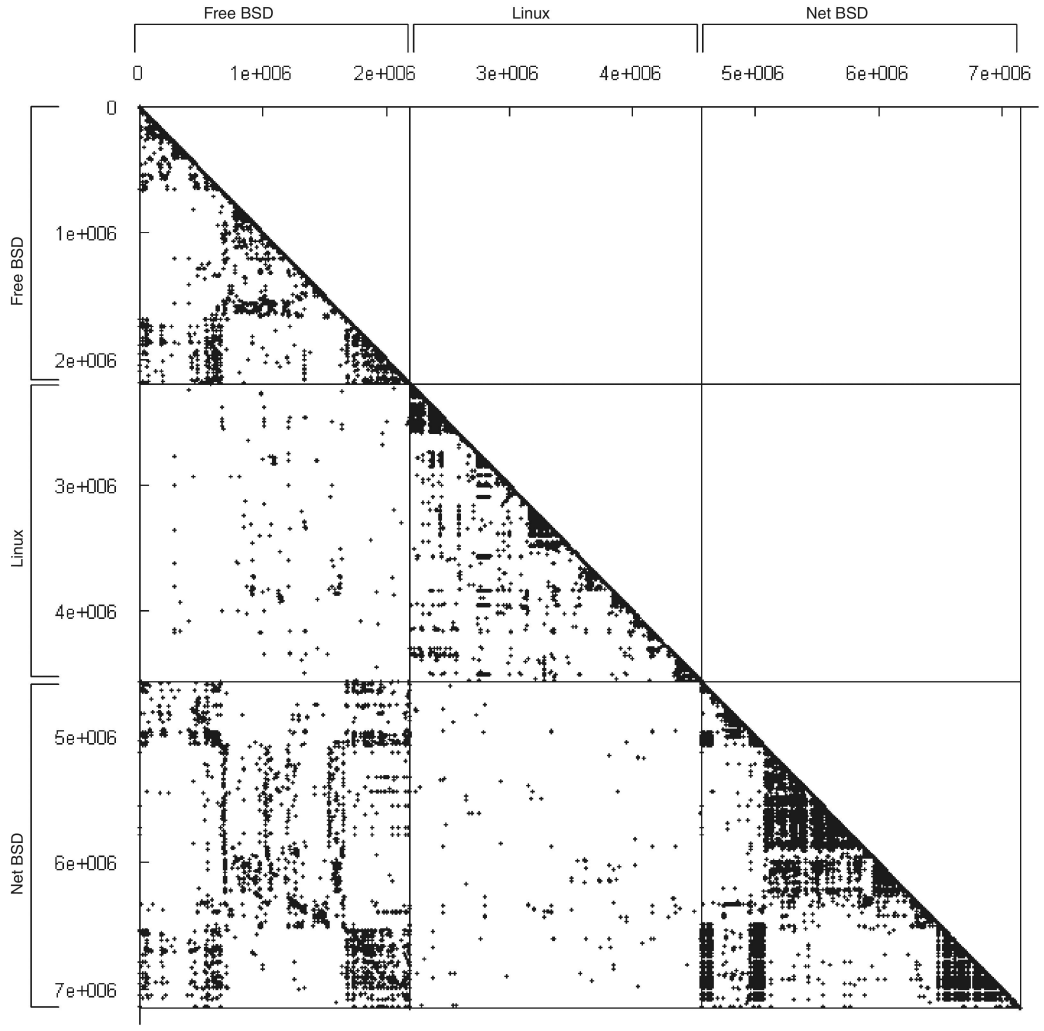


Fig. 14. Scatter plot of clones over 30 token among three operating systems.

Each OS has many clones in itself and, also, a large number of clones have been found between FreeBSD and NetBSD. On the other hand, a small number of clones were found between Linux and the other two. Table 4 shows the number of clone pairs and the COVERAGE between the three systems. The situation of clones is confirmed again, quantatively. About 40 percent of source files in FreeBSD have clones with NetBSD; whereas, less than 5 percent of source files in FreeBSD or NetBSD have clones with Linux.

To investigate what type of similar code portions were used between FreeBSD and Linux, CCFinder was applied only to source files that have clones with the other OS.

TABLE 4
Clones between OSs

	Clone pairs	COVERAGE (% LOC)	COVERAGE (% FILE)
FreeBSD and Linux	1091	0.8% of FreeBSD	3.1% of FreeBSD
		0.9% of Linux	4.6% of Linux
FreeBSD and NetBSD	25621	18.6% of FreeBSD	40.1% of FreeBSD
		15.2% of NetBSD	36.1% of NetBSD
Linux and NetBSD	1000	0.6% of Linux	3.3% of Linux
		0.6% of NetBSD	2.1% of NetBSD

Those are 149 files of 205K lines in FreeBSD and 149 files of 281K lines in Linux. The scatter plot is shown in Fig. 15 and the source files of the major clone groups marked $A, B, \ldots I$ are summarized in Table 5. For clone pairs at *A*, *C*, and *G*, source files of each pair have the same name. Almost all clone pairs at *I* have source files of the same name. These clones therefore could be found by comparing file names, without using CCFinder. On the other hand, clone pairs at *E* and *F* have different file names for different systems, so that it is more difficult to find these without the tool.

A more complex situation arises among clone pairs of *B*, *D*, and *J*. All files related to *B*, *D*, and *J* are further investigated. FreeBSD has four source files that have clones with *B*, *D*, or *J*. Linux has nine such files. Fig. 16 shows a scatter plot of clone pairs among these files. *P* shows that two files with different names, `sys/net/zlib.c` of FreeBSD and `drivers/net/zlib.c` of Linux are almost identical. Also, in Linux, the smaller zlib.c (`arch/ppc/coffboot/zlib.c`) is a subset of the larger zlib.c (`driver/net/zlib.c`), as shown by *Q*. Moreover, the

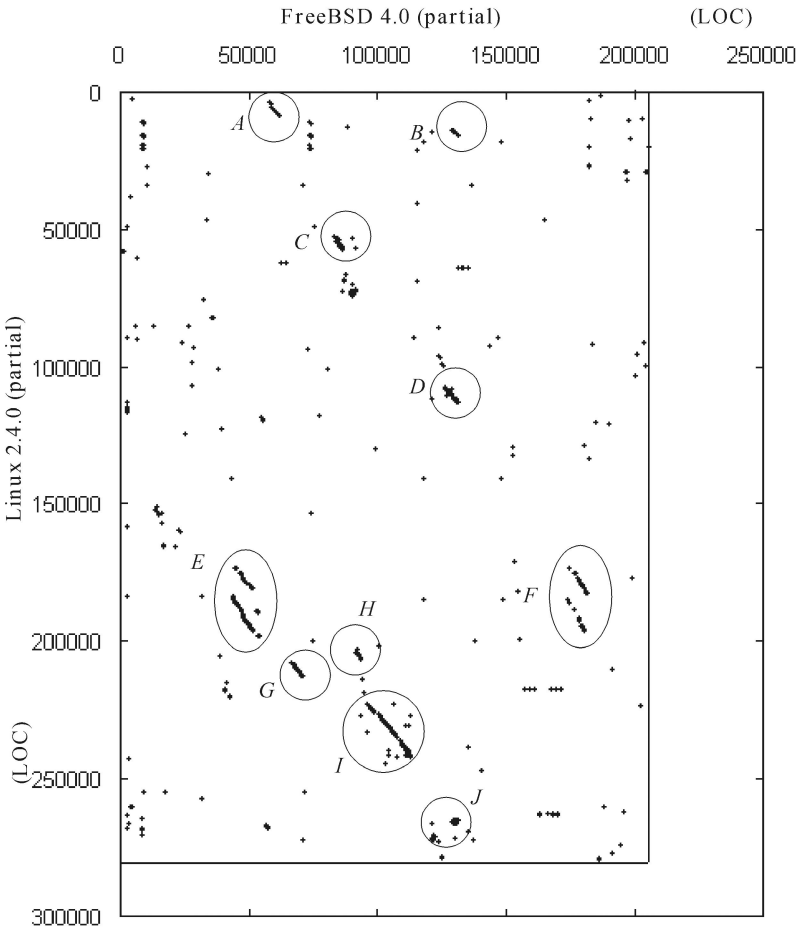


Fig. 15. Scatter plot of clones between FreeBSD and Linux found in source files only with clones.

TABLE 5
Source Files Including Those Marked in Fig. 15

Group	FreeBSD	Linux
A	`sys/gnu/i386/fpemul/*.c` (6 files)	`arch/i386/math-emu/*.c` (6 files)

B, D, J	`sys/net/zlib.c`	`arch/ppc/coffboot/zlib.c`
		`drivers/net/zlib.c`
		`fs/cramfs/inflate/*.c` (8 files)

C	`sys/i386/isa/istallion.c`	`drivers/char/istallion.c`
E, F	`sys/dev/sym/sym_hipd.c` `sys/pci/{ncr.c, scsiiom.c}`	`drivers/scsi/{ncr53c8xx.c, sym53c8xx.c, scsiiom.c}`
G	`sys/gnu/i386/isa/sound/awe_wave.c`	`drivers/sound/awe_wave.c`
H	`sys/i386/isa/sound/{ad1848.c, audio.c}`	`drivers/sound/{ad1816.c, ad1848.c, audio.c }`
I	`sys/i386/isa/sound/*.c` (23 files)	`drivers/sound/*.c` (22 files)

smaller zlib.c contains `fs/cramfs/inflate/*.c` files, as shown by *R*.

3.3 Other Case Studies

3.3.1 C Program

CCFinder was used to compare C source files of two systems developed by two companies, *A* and *B*. Company *A* filed a lawsuit against company *B* since the *B*'s system was suspected to be an illegal copy of *A*'s system. Each system consists of more than 100 files written in C and COVERAGE values are over 50 (% FILE). The result was filed in a court as an evidence document and this matter is still awaiting the Court's decision.

3.3.2 COBOL Program

CCFinder is also being applied to a huge government system consisting of about 1 million lines in 2,000 modules, which was also the initial motivation of this research. It is written in COBOL and a PL/I-like language and it has been used over 20 years and maintained continuously by many engineers. Our first analysis result revealed that half of the pairs of modules with intermodule clones have small age gap ($\leq$ 30 days). Also, the module that have relatively long code clones tends to have faults [18].

4 Related Works and Discussion

Here, we compare our proposed clone detection method (CCFinder) with other methods from various viewpoints.

4.1 Existing Clone Detection Approaches and Tools

A clone detection tool **Dup** [1] uses a sequence of lines as a representation of source code and detects line-by-line clones. It performs the following subprocesses: 1) replacement of identifiers of functions, variables, and types into a special identifier (parameter identifier), 2) extraction of matches by a suffix-tree algorithm [10] of O(n) time complexity (n is the number of lines in the input), 3) computation of correspondence (pairing) between parameter identifiers. The line-by-line method has a weakness in the line-structure modification, which will be discussed in Section 4.2.

A language dependent clone detection tool **Duploc** [7] reads source files, makes a sequence of lines, removes white-spaces and comments in lines, and detects match by a string-based Dynamic Pattern Matching (DPM). The output is the line numbers of clone pairs, possibly with gap (deleted) lines in them. The computation complexity is $O(n^2)$ for the input size n and it is practically too expensive. The tool uses an optimization technique by a hash function for string, which reduces the computation complexity by the factor *B*, which is a constant determined by the number of characters in a line.

Baxter et al. proposes a technique to extract clone pairs of statements, declarations, or sequences of them from C source files [5]. The tool parses source code to build an abstract syntax tree (AST) and compares its subtrees by characterization metrics (hash functions). The parser needs a "full-fledged" syntax analysis for C to build AST. Baxter's tool expands C macros (define, include, etc) to compare code portions written with macros. Its computation complexity is $O(n)$, where n is the number of the subtree of the source files. The hash function enables one to do parameterized matching, to detect gapped clones, and to identify clones of code portions in which some statements are reordered. In AST approaches, it is able to transform the source tree to a regular form as we do in the transformation rules. However, the AST based transformation is generally expensive since it requires full syntax analysis and transformation.

A clone-detecting method proposed by Mayrand et al. uses a representation named Intermediate Representation Language (IRL) to characterize each function in the source code [17]. A clone is defined only as a pair of whole function bodies that have similar metric values.

A clone detection tool **SMC** (Similar method Classifier) [3] uses a hybrid approach of characterization metrics and DPM (dynamic pattern matching). The paper only discusses detection of whole methods, although the approach would be applied to detect partial code portions also. The detection process consists of the following subprocesses:

1. extraction of method bodies from source code of Java,
2. computing characteristic metrics values for each method,
3. identify pairs of methods with similar metric values,
4. compare each pair of token sequences of the similar methods by DPM to identify clone methods, and
5. classifying clone methods into 18 categories.

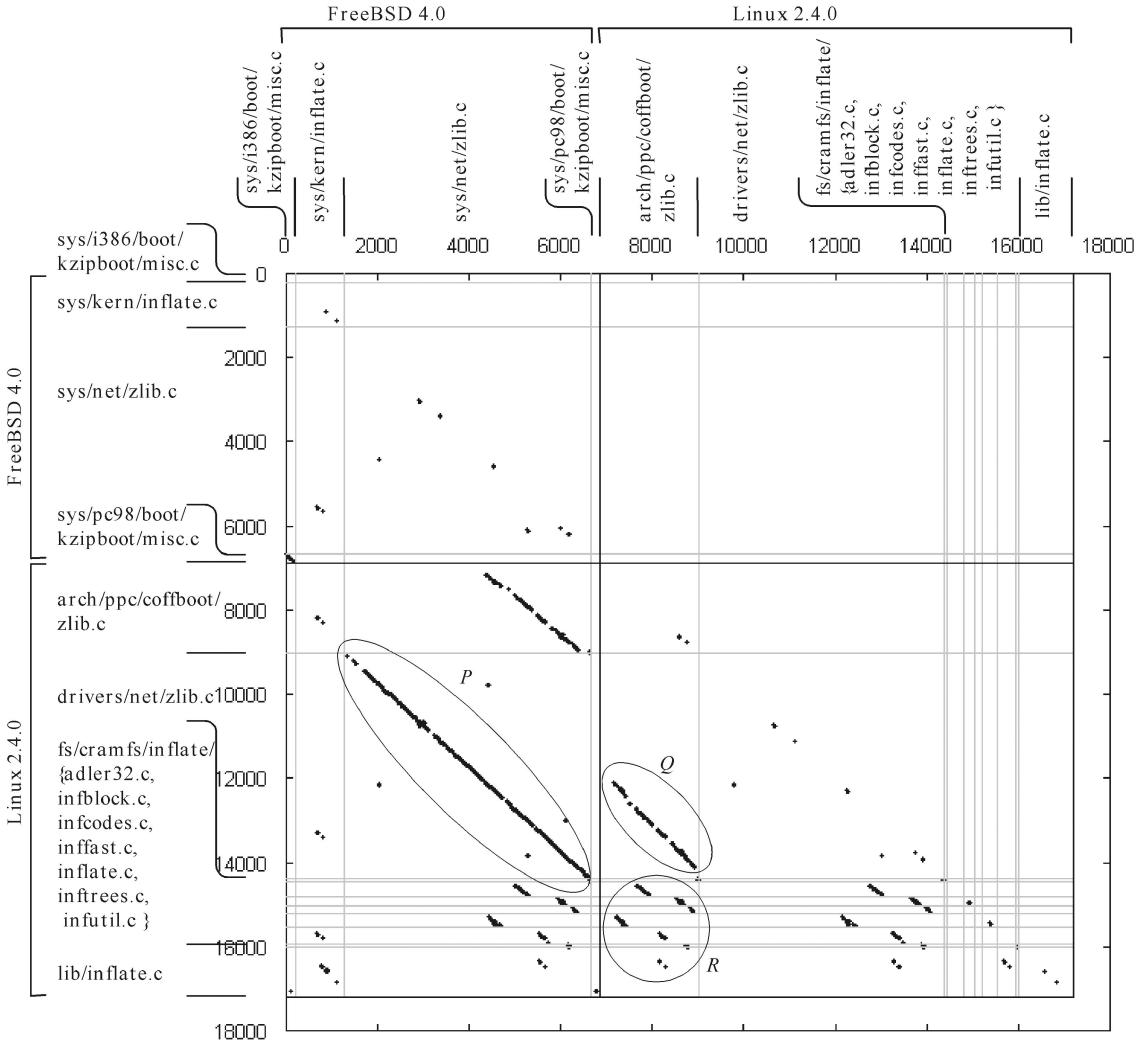


Fig. 16. Clones among zlib subsystems.

The computing complexity is O(n), where n is amount of methods in source files. The tool might be easily ported to the languages to which the metrics are applicable, although its parser and hash function have to be constructed and tuned for the input language.

4.2 Discussions of Clone Detection Approach

4.2.1 Granularity in Representation of Source Code

Clone code detection methods are classified by the granularity of internal representation of source code. The approaches described in Section 4.1 use one of the following data representational granularity.

- Functions and methods (Mayrand's [17], SMC [3]).
- Lines (Dup [1]).
- Nodes in AST (Abstract Syntax Tree) (Baxter's [5]).
- Tokens (CCFinder).
- Characters (Duploc [7]).

When a source code is internally represented as a sequence of lines, a matching algorithm compares code portions line-by-line. Since in "free-format" languages such as C, C++, and Java, line breaks in source code have no semantic meaning, they are often placed and relocated based on developer's preference. A weakness of line-by-line comparison is that code portions with such line break relocation are not detected as clones or detected as shorter clones.

Fig. 17 shows an example found in the case study of FreeBSD and Linux described in Section 0. The line break at line 343 is relocated up to the end of previous line in the latter case (line 785), so the usual line-by-line method cannot recognize it. In this case study, 252 clone pairs of detected 1,091 clone pairs (23 percent) have the line break relocation and they are not detected by usual line-by-line method. Another issue for the line-based representation is that it is not suitable for transformation as we do in CCFinder. In this case study, CCFinder is wise enough to detect whole of these code portion as clones. The differences are not only the line break relocation but also name changes (e.g., `dwFrameGroupLength` at 341 → `frameGroupLength` at 784) and parenthesis removal for a single statement in a compound block (346 and 348). It is another coding style whether or not a single statement is surrounded by "{" and "}" as a compound block just after "`if(...)`," "`else`," or "`for(...)`." In the case study, a total of 361 (33 percent) clone pairs have either the line break relocation or the parenthesis removal, and they cannot be detected ordinary tools.

```
341:    dwFrameGroupLength = 1;
342:    for(dwCnt = 2; dwCnt <= 64; dwCnt *= 2)
343:    {
344:            if(((ulOutRate / dwCnt) * dwCnt) !=
345:               ulOutRate)
346:            {
347:                    dwFrameGroupLength *= 2;
348:            }
349:    }
```

```
784:    frameGroupLength = 1;
785:    for (cnt = 2; cnt <= 64; cnt *= 2) {
786:            if (((rate / cnt) * cnt) != rate)
787:                    frameGroupLength *= 2;
788:    }
```

Fig. 17. Part of code portions of a clone pair found between FreeBSD and Linux.

Since a source line generally contains several tokens, token-based representation requires longer CPU time and larger memory than line-by-line comparison. However, token-based representation can easily employ various transformations, by which difference of coding style is effectively eliminated and many code portions are detected as clone pairs.

In AST-based representation, subtrees of the AST are candidates of clone. The comparison requires type information, e.g., two variables are equivalent if they have a same type. Analysis of type information can be obtained by similar process as front end of compilers, which can be generally obtained by comprehensive procedures.

An advantage of the character-based representation is high adaptability to various programming languages since no lexical analysis or parsing is needed. For example, tool Duploc [7] can detect clones in source files written in C, Smalltalk, Python, and COBOL. However, the character-based representation is generally expensive and it requires elaboration to have tools scalable.

4.2.2 Matching Algorithm

Another viewpoint to classify clone detection methods is the matching algorithm to detect clones.

- Suffix-tree algorithm (Baker [1] and CCFinder).
- DPM(Dynamic Pattern Matching) (Duploc [7] and Kontogiannis's [14]).
- Hash-value comparison (Baxter's [5], Mayrand's [17], and SMC [3]).

Suffix-tree algorithm computes all same subsequences in a sequence composed of a fixed alphabet (e.g., characters, tokens, or hash values of lines). The computation complexity is $O(n)$, here n is the length of the input sequence.

DPM computes a longest common subsequence between two sequences. The computation complexity is $O(mn)$ in a naive method; here, m and n are the length of the sequences. Consequently, it has $O(n^2)$ complexity in order to compute matches inside a single sequence with length n. Two optimization techniques used to reduce the complexity are proposed in [3], in which a limitation of gap length is employed and characteristic metrics for a preliminary selection is used. However, the scalability of the approach to millions of line is still unknown.

The matching algorithm with hash-value comparison first computes metric values from code portions. Then, the metric values are ordered and code portions with similar metric values are identified. The time and space complexity is $O(n)$ for size n input. Metrics and the ordering method heavily depend on the input language. Also, the granularity of the compared code portions would affect the applicability of the metrics and they affect effectiveness and precision. Thus, tuning to an input language is generally difficult and porting to a different input language is not an easy task.

4.3 Methods and Techniques Around Clone Detection

4.3.1 Gapped Clone Detection

The basic idea of CCFinder is to transform a token sequence in a regularized form on which two code portions likely to be cloned are identical sequences of tokens. Source code is represented as a sequence of tokens for the need of transformation rules. The transformations insert or delete tokens in a sequence; thus, a kind of gapped clones is detected as a clone by the transformation rules. Some gapped clone detection methods described above use nongapped clones as a elements of a gapped one, or use the length of found nongapped clones as a threshold of the gap size. We can employ these approaches to CCFinder to find various kinds of gapped clones.

4.3.2 Merging Clone Code Portions

How to rewrite each detected clone code as a shared code depends on programming language. Baker proposes a rewriting method of clone pairs in C source as a shared code with C macro [1]. Baxter et al. have built a tool that generates such macros [5]. Their tools determine correspondence of identifiers between two code portions of a clone pair.

Balazinska et al. developed redesign patterns for Java [4]. Their approach is at first to classify clone pairs of Java methods by the difference (variables, constants, expressions, signature of methods, or calling methods) and then apply the redesign pattern for each classification.

The target of these writing approaches is a clone pair. Merging a clone class with more than two code portions would be a more challenging, although it is practically important task.

4.3.3 Clone Analysis over Versions

In [15], Laguë et al. performed tracking of clones over versions of a system, using the clonedetection technique presented in [17]. In [12], Johnson examined changes between two versions of a compiler *gcc* [9], with clone pairs composed sequence of lines.

5 Conclusions and Future Work

In this paper, we have presented a clone detecting technique with transformation rules and a token-based comparison, as well as important optimization techniques to improve performance and efficiency. We have also proposed metrics to select interesting clones. They have been applied to several industrial-size software systems in the case studies. A case study of JDK has shown that the transformation rules support to extract certain kinds of clones. Clone metrics have been used to investigate the code clones and the source files. In another case study of three operating systems, FreeBSD, Linux, and NetBSD, the clone analysis has revealed that FreeBSD and NetBSD have similar source codes and that these two are dissimilar to Linux, in quantitative way. The detailed investigation about zlib.c has turned out that each of Linux and FreeBSD has several source files that have duplicated codes (or subset) of zlib.c.

Our current clone detection tool does not accept source files written in two or more programming languages. However, today some software systems are implemented in multilanguages (e.g., C and C++, Java and HTML, and etc.). We are trying to extend the tool to accept source programs written in several programming languages at the same time.

Acknowledgments

The authors would like to thank Professor Hideo Matsuda of Osaka University for his helpful comments on the suffix tree algorithm.

References

[1] B.S. Baker, "A Program for Identifying Duplicated Code," *Proc. Computing Science and Statistics: 24th Symp. Interface,* vol. 24, pp. 49-57, Mar. 1992.
[2] B.S. Baker, "On Finding Duplication and Near-Duplication in Large Software System," *Proc. Second IEEE Working Conf. Reverse Eng.,* pp. 86-95, July 1995.
[3] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K.A. Kontogiannis, "Measuring Clone Based Reengineering Opportunities," *Proc. Sixth IEEE Int'l Symp. Software Metrics (METRICS '99),* pp. 292-303, Nov. 1999.
[4] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K.A. Kontogiannis, "Partial Redesign of Java Software Systems Based on Clone Analysis," *Proc. Sixth IEEE Working Conf. Reverse Eng. (WCRE '99),* pp. 326-336, Oct. 1999.
[5] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," *Proc. IEEE Int'l Conf. Software Maintenance (ICSM '98),* pp. 368-377, Nov. 1998.
[6] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler, "GJ Specification," http://cm.bell-labs.com/cm/cs/who/wadler/pizza/gj/, 1998.
[7] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code," *Proc. IEEE Int'l Conf. Software Maintenance (ICSM '99),* pp. 109-118, Aug. 1999.
[8] *FreeBSD,* http://www.freebsd.org/, 2002.
[9] *Gnu Project,* http://www.gnu.org/, 2002.
[10] D. Gusfield, *Algorithms on Strings, Trees, and Sequences.* pp. 89-180, Cambridge University Press, 1997.
[11] J.H. Johnson, "Identifying Redundancy in Source Code Using Fingerprints," *Proc. IBM Centre for Advanced Studies Conference (CASCON '93),* pp. 171-183, Oct. 1993.
[12] J.H. Johnson, "Substring Matching for Clone Detection and Change Tracking," *Proc. IEEE Int'l Conf. Software Maintenance (ICSM '94),* pp. 120-126, Sept. 1994.
[13] B.-K. Kang and J.M. Bieman, "Using Design Abstractions to Visualize, Quantify, and Restructure Software," *J. Systems and Software,* vol. 24, no. 2, pp. 175-187, 1998.
[14] K.A. Kontogiannis, R. De Mori, E. Merlo, M. Galler, and M. Bernstein, "Pattern Matching Techniques for Clone Detection and Concept Detection," *J. Automated Software Eng.,* vol. 3, pp. 770-108, 1996.
[15] B. Laguë, E.M. Merlo, J. Mayrand, and J. Hudepohl, "Assessing the Benefits of Incorporating Function Clone Detection in a Development Process," *Proc. IEEE Int'l Conf. Software Maintenance (ICSM '97),* pp. 314-321, Oct. 1997.
[16] *Linux Online,* http://www.linux.org/, 2002.
[17] J. Mayrand, C. Leblanc, and E.M. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics," *Proc. IEEE Int'l Conf. Software Maintenance (ICSM '96),* pp. 244-253, Nov. 1996.
[18] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto, "Software Quality Analysis by Code in Industrial Legacy Software," Proc. IEEE Eighth Int'l Software Metrics Symp. (METRICS '02), (to appear) June 2002.
[19] *NetBSD Project,* http://www.netbsd.org/, 2002.
[20] *OpenOffice.org Source Project,* http://www.openoffice.org/, 2002.
[21] S. Takabayashi, A. Monden, S. Sato, K. Matsumoto, K. Inoue, and K. Torii, "The Detection of Fault-Prone Program Using a NeuralNetwork," *Proc. SEA-UNU/IIST Int'l Symp. Future Software Technology (ISFST '99),* pp. 81-86, Oct. 1999.
[22] *The Source for Java Technology,* http://java.sun.com/, 2002.

Toshihiro Kamiya received the ME and PhD degrees in information and computer science from Osaka University in 1999 and 2001, respectively. He is a research associate of Intelligent Cooperation and Control Group, RPESTO, Japan Science and Technology Corporation, and dispatched to Nara Institute of Science and Technology. His research interests are in the areas of software metrics and cognitive science. He is a member of the IPSJ, the IEEE, and the IEEE Computer Society.

Shinji Kusumoto received the BE, ME, and DE degrees in information and computer sciences from Osaka University in 1988, 1990, and 1993, respectively. He is currently an associate professor at Osaka University. His research interests include software metrics and software quality assurance technique. He is a member of the IEEE, the IEEE Computer Society, IPSJ, IEICE, and JFPUG.

Katsuro Inoue received the BE, ME, and PhD degrees in information and computer sciences from Osaka University, Japan, in 1979, 1981, and 1984, respectively. He was an assistant professor at the University of Hawaii at Manoa-from 1984-1986. He was a research associate at Osaka University from 1984-1989, an assistant professor from 1989-1995, and a professor from 1995. His interests are in various topics of software engineering such as software process modeling, program analysis, and software development environment. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

▷ **For more information on this or any computing topic, please visit our Digital Library at** http://computer.org/publications/dilb.