Tree-Pattern-Based Clone Detection with High Precision and Recall

Hyo-Sub Lee[1], Myung-Ryul Choi[2] and Kyung-Goo Doh[3]
[1] Department of Computer Science, College of Engineering, Hanyang University
Seoul, 04763 – Republic of Korea
[e-mail: hyosub@hanyang.ac.kr]
[2] Division of Electronic Engineering, College of Engineering, Hanyang University ERICA
Ansan, 15588 – Republic of Korea
[e-mail: choimy@hanyang.ac.kr]
[3] Division of Computer Science, College of Computing, Hanyang University ERICA
Ansan, 15588 – Republic of Korea
[e-mail: doh@hanyang.ac.kr]
*Corresponding author: Kyung-Goo Doh

Received April 15, 2017; revised July 27, 2017; accepted December 10, 2017; published May 31, 2018

Abstract

The paper proposes a code-clone detection method that gives the highest possible precision and recall, without giving much attention to efficiency and scalability. The goal is to automatically create a reliable reference corpus that can be used as a basis for evaluating the precision and recall of clone detection tools. The algorithm takes an abstract-syntax-tree representation of source code and thoroughly examines every possible pair of all duplicate tree patterns in the tree, while avoiding unnecessary and duplicated comparisons wherever possible. The largest possible duplicate patterns are then collected in the set of pattern clusters that are used to identify code clones. The method is implemented and evaluated for a standard set of open-source Java applications. The experimental result shows very high precision and recall. False-negative clones missed by our method are all non-contiguous clones. Finally, the concept of neighbor patterns, which can be used to improve recall by detecting non-contiguous clones and intertwined clones, is proposed.

Keywords: Software maintenance, code clone, clone detection, abstract syntax tree

http://doi.org/10.3837/tiis.2018.05.002 **ISSN : 1976-7277**

1. Introduction

Code clones are a set of code fragments that are identical or similar to one another. Identifying clones in source code is known to be very important in software maintenance. Because the inspection of code clones with naked eyes can be time-consuming and inconsistent, the use of automatic means is preferred. However, the clone-detection problem is computationally expensive to solve, and thus in practice settles for approximate solutions, inevitably including false positives (due to lack of precision) and false negatives (due to lack of recall) [1,2,3,4]. This paper proposes a code-clone detection method that gives the highest possible precision and recall, without giving much attention to efficiency and scalability. The goal is to automatically create a reliable reference corpus that can be used as a basis for evaluating the precision and recall of clone detection tools.

Tree-based clone detection naturally gives better precision than its token-based counterpart because it possesses extra structural information [6,7]. However, when it uses complete subtree as comparison unit, it is probable that two almost identical trees are not detected as clones, resulting in false negatives and thus suffering loss of recall. For example, consider two small Java code fragments shown in **Fig. 1(1)** picked up from an open-source real-world Java application. A tree-based tool, CloneDR® [6], only detects subtrees of if-statements (framed by solid lines in **Fig. 1(2)**) as clones[1]. However, if you look closely at either codes or trees, you obviously want to insist that the two entire for-loops be bigger clones. Except small subtree pairs of①'s, ②'s and ③'s circled by dotted lines in **Fig. 1(2)**, the two trees are indeed entirely identical. Thus the two code fragments in **Fig. 1(1)** should be detected as clones.

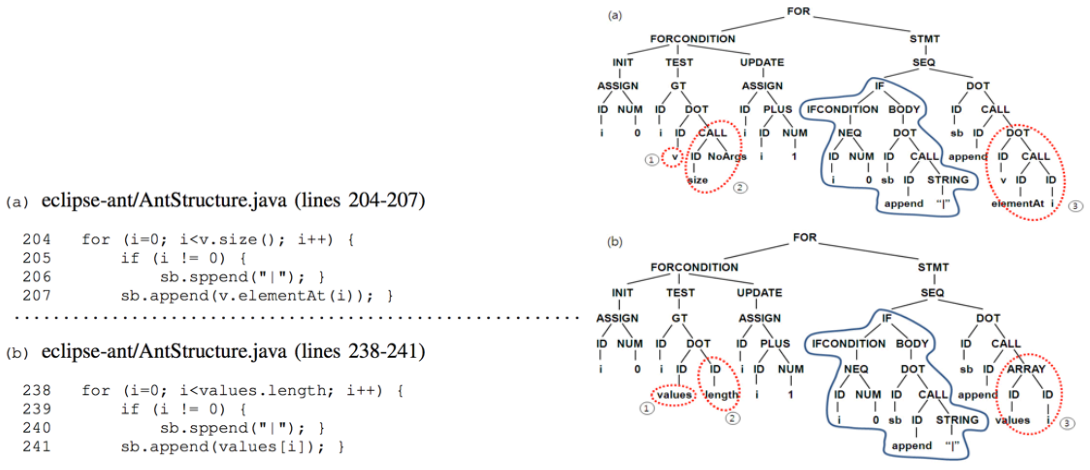


Fig. 1. Example: false-negative clones

Some tree-based clone detection tools employ the method of partially abstracting away syntactic structural information in favor of improving efficiency and scalability. When a clone detection tool compares the abstracted representations of trees, it is possible that two completely different trees are detected as clones, resulting in false positives and thus suffering loss of precision. For examples, DECKARD compares characteristic-vector representations of

[1] Note that we ignore clones with the number of nodes less than 10.

trees and is thus fast and scalable [5]. The characteristic vector captures a tree structure as a one-dimensional vector of natural numbers, abstracting away some structural information. **Fig. 2** shows an example of a false-positive clone accidentally detected. Their characteristic vectors are identical, but the two code fragments are completely different.

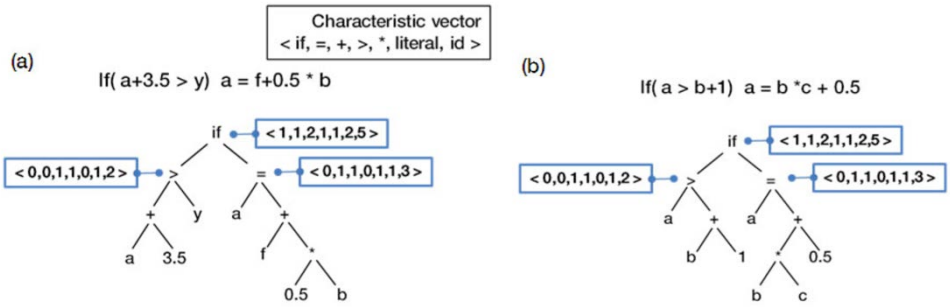


Fig. 2. Example: false-positive clone

The best way to avoid false positives is to keep the trees intact, bearing the computational overhead. To remove false negatives, Evans-Fraser-Ma first suggested finding duplicate tree patterns rather than duplicate complete subtrees [8]. A tree pattern is essentially a tree except that its leaf may be a hole, a special node that can be replaced by the root of any tree with no hole. **Fig. 3** shows the duplicate tree patterns (framed in a solid line) of two ASTs of code fragments in **Fig. 1(1)**. In each duplicate tree pattern, there are three holes.

This paper introduces a tree-pattern-based code clone detection that shows very high precision and recall. Program trees (i.e., abstract syntax trees) are compared intact in order to keep precision as high as possible. The comparison unit is a tree pattern – a tree with holes and gaps – rather than a complete subtree in order to keep recall as high as possible. For each subtree pair, the algorithm traverses both trees top-down and in breadth-first manner, and collects the biggest possible tree patterns. The algorithm is designed in such a way that the same comparison is never repeated.

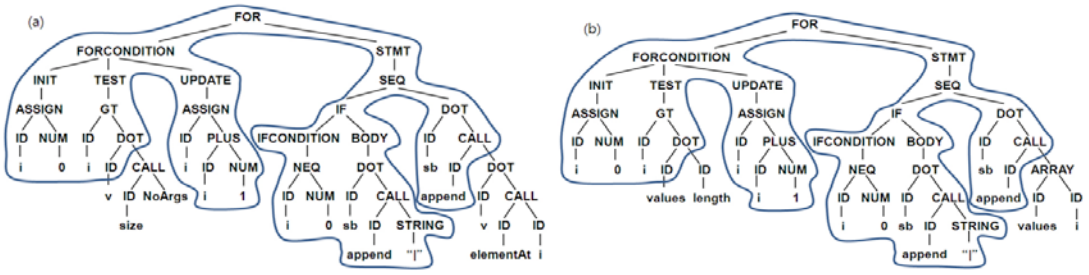


Fig. 3. Duplicate tree patterns

The contributions of this paper are as follows:

- We developed a top-down algorithm for detecting duplicate tree patterns in two given program trees based on anti-unification. The comparisons are exhaustive, but redundancy free. False positives occur less since trees are compared intact. False negatives occur less since tree patterns are compared.
- We implemented the tree-pattern-based clone detector, CCR(Code Clone Detector), and showed that the tool has very high precision and recall through experiments using standard open-source Java applications. The manually created Bellon's reference

corpus is used to carry out the evaluation of recall. Clone pairs of type 1 and 2 in the reference corpus are completely detected, and those of type 3 are all detected except non-contiguous clones. Compared to Evans-Fraser-Ma's bottom-up algorithm, ours shows twice as much better recall than theirs. Moreover, flaws presented in **Fig. 1** and **Fig. 2** are all remedied.

- We proposed the concept of neighbor patterns to improve recall that would help detect non-contiguous clones and interwined clones.

The remainder of the paper is organized as follows. Section 2 discusses the terminologies used in the paper. Section 3 describes an algorithm for detecting duplicate tree pattern clusters based on anti-unification and clustering. Section 4 discusses in detail the implementation of CCR and evaluates its precision and recall. Section 5 proposes how to detect non-contiguous clones and intertwined clones by explaining the concept of neighbor patterns. Section 6 surveys the existing techniques of automatic clone detection. Finally, section 7 concludes with possible future research.

2. Preliminaries

A *ranked alphabet* is a tuple (S,α) where S is a finite set of symbols, α is a mapping from each symbol in S to its arity represented by a natural number greater than or equal to 0. That is, the arity of a symbol $s \in S$ is $\alpha(s)$. The set of symbols of arity k is denoted by S_k. That is, S_0 is the set of constant symbols, S_1 is the set of unary symbols, . . . , S_k is the set of k-ary symbols. We assume that S contains at least one constant.

A *tree* is represented as a term in a ranked alphabet defined as follows: the set $T(S)$ of trees over the ranked alphabet S is the smallest set inductively defined by:

1) $S_0 \subseteq T(S)$
2) For $k \geq 1$, if $s \in S_k$ and $\forall t_{i:1 \leq i \leq k} \in T(S)$, then $s(t_1, \ldots, t_k) \in T(S)$

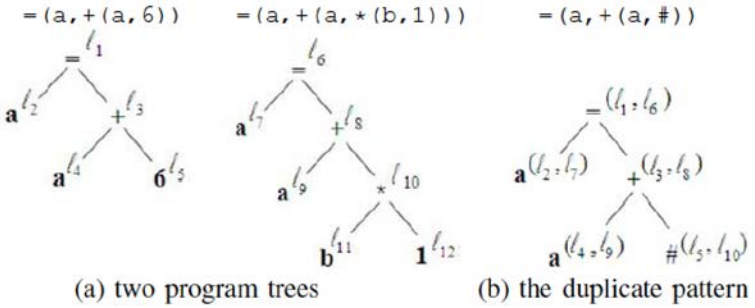
If a tree t has the form of $s(t_1, \ldots, t_k)$, we say that the function symbol s is a root node of t , and $t_1, \ldots, t_k$ are immediate sub-trees of t. The *size* of a tree is the number of symbols (in fact, nodes) in the tree.

The most specific generalization (MSG) of two trees is the largest connected common component of two trees, sharing their root. For example, suppose we have two trees, `=(a,+(a,6))` and `=(a,+(a,*(b,1)))`. The MSG of the two is: `=(a,+(a,#))`, where a special character `'#'` (called *hole*) replaces uncommon sub-trees, `6` and `*(b,1)`, in each of the original trees. We assume that `'#'` is not in the original set of symbols. Note that the MSG of two trees with different root symbols is `'#'` and that of two identical trees has no hole. Generally speaking, the MSG of two trees becomes a tree pattern that is a tree having zero or more holes as some of its leaf nodes.

A *program tree*, which is the abstract-syntax tree (AST) representation of a program, is a labeled tree in which each node is decorated with a unique label that represents its corresponding program point in the source program text, as shown in **Fig. 4(a)**. A program tree with no label is called a *skeleton*. A *duplicate pattern* is the MSG of two program trees that can be obtained by constructing the MSG of their skeletons and decorating each node with a pair of labels from its corresponding original node, as shown in **Fig. 4(b)**.

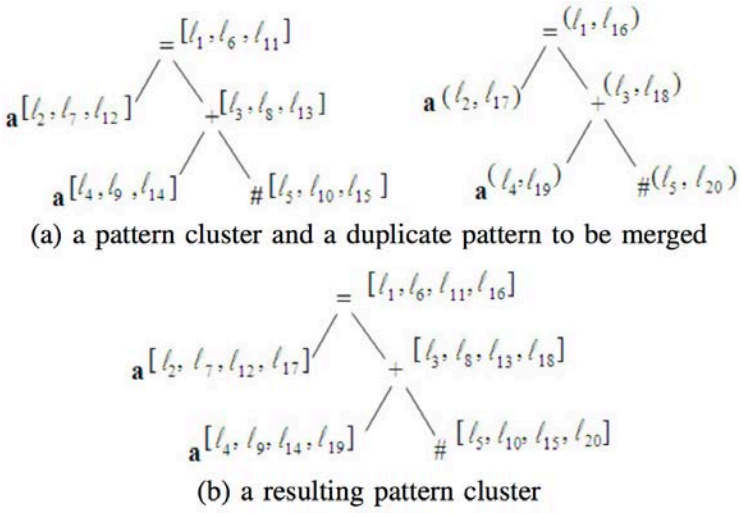
Duplicate patterns of identical skeletons can be merged to become a *pattern cluster*. Each node of a pattern cluster is decorated with a list of labels. When a duplicate pattern is merged into a pattern cluster, the label pair of each node in the duplicate pattern is appended to the

label list of the corresponding node in the pattern cluster. The label that is already in the label list is not appended. Note that the length of each label list in a pattern cluster is always identical because every label in the list is unique. **Fig. 5** shows how a duplicate pattern is merged into a pattern cluster.



(a) two program trees (b) the duplicate pattern

Fig. 4. Example: the duplicate pattern of two program trees



(a) a pattern cluster and a duplicate pattern to be merged

(b) a resulting pattern cluster

Fig. 5. Example: a duplicate pattern is merged into a pattern cluster

3. Algorithm

In this section, we present an algorithm that, given two program trees, finds all pattern clusters by comparing every pair of sub-trees in the two trees. The algorithm is designed in such a way that only the most general (the largest possible) duplicate pattern is kept. The main algorithm is shown in **Fig. 6** [9].

First, all the largest possible duplicate patterns are searched and collected. The goal is to exhaustively compare every pair of sub-trees, achieving high precision and recall, while avoiding the repetition of same comparisons, reducing its computational cost as much as possible. To avoid the repetition, a two-dimensional array, *work-table*, is maintained to record which pair of nodes were already compared. A pair of sub-trees is compared starting from the roots in breadth-first order, and a duplicate pattern is constructed in top-down fashion from its root. If two nodes compared are identical, then the duplicate pattern grows downwards with the node decorated with a pair of labels taken from the two original nodes. Otherwise, the duplicate pattern construction stops by replacing the node with a special leaf node, named *hole*.

This process continues until every leaf in the pattern is either the copy of a leaf from original sub-trees or a hole. An already compared pair of nodes is marked as such in work-table. Once marked, two sub-trees whose root nodes are such a pair need not be compared again, and thus removed from the list of sub-trees to compare. Identified patterns are collected into a map indexed by its root node. This process specified as *anti-unify* function in **Fig. 7** resembles the anti-unification algorithm first developed by Plotkin [10] and Reynolds [11] in the early 1970s.

```
1   algorithm pattern-collection(t1, t2:tree)
2     returns clustered-patterns
3   type
4     work-table = array [L1][L2] of boolean ;
5           where Li is the set of labels in ti, i = 1 or 2
6     patterns = map of S to 2^P ;
7     clustered-patterns = map of S to 2^C ;
8           where S is the set of function symbols of arity ≥ 1
9             and P is the set of patterns
10            and C is the set of clustered patterns
11  var
12    worklist : work-table ;
13    pats : patterns ;
14    cpats : cluster-patterns ;
15  begin
16    ∀l1 ∈ L1, ∀l2 ∈ L2, worklist[l1][l2] ← true ;
17    ∀s ∈ S, pats[s] ← ∅ ;
18    ∀s ∈ S, cpats[s] ← ∅ ;
19    for all subtrees t'1 of t1 and t'2 of t2 do
20              where t'1 = s1^l1(u1, ..., um), m ≥ 0
21                and t'2 = s2^l2(v1, ..., vn), n ≥ 0
22      if worklist [l1][l2] then
23          worklist [l1][l2] ← false ;
24          if s1 = s2 then
25              pats[s1] ← pats[s1] ∪ anti-unify(t'1, t'2, worklist) ;
26    for all s ∈ S do
27      cpats[s] ← clustering(pats[s]) ;
28    return cpats ;
29  end
```

Fig. 6. Algorithm: Duplicate Pattern Collection

Next, once all duplicate patterns are collected, they are clustered into a pattern cluster by *clustering* function described in **Fig. 8**. Duplicated patterns with identical skeleton are all clustered into a pattern cluster.

```
1   function anti-unify (t1, t2 : tree, var worklist : work-table)
2                         where t1 = s1^l1(u1, ..., um), m ≥ 0
3                           and t2 = s2^l2(v1, ..., un), n ≥ 0
4     returns tree-patterns
5   begin
6     worklist[l1][l2] ← false ;
7     if s1 ≠ s2 then return #^(l1,l2) ;
8     else (* s1 = s2 and m = n in AST *)
9       if m = n = 0 then (* t1 and t2 are leaves *)
10        return s1^(l1,l2) ;
11      else (* m = n ≠ 0 *)
12        return s^(l1,l2) (anti-unify (u1, v1, worklist), ... ,
13                          anti-unify (um, vn, worklist)) ;
14  end
```

Fig. 7. Anti-unification

```
1   function clustering (ps : set of patterns)
2     returns set of clustered-patterns
3   var
4     merged : boolean ;
5     p : patterns ;
6     c : clustered-patterns ;
7     cs : set of clustered-patterns ;
8   begin
9     cs ← { };
10    for each p in ps do
11      merged ← false ;
12      for each c in cs do
13        if p and c are the same shape then
14          merge labels of p into c ;
15          merged ← true ;
16          break out of inner for-loop ;
17      if not merged then
18        cs ← cs ∪ {p};
19    return cs ;
20  end
```

Fig. 8. Pattern Clustering

Each node in a pattern cluster has the list of labels representing its corresponding program points in original program. In addition, every hole in a pattern cluster records the size of each sub-tree originally hanged on the hole. We call this value *hole mass* that is to be used as a reference value when deciding the enclosing pattern cluster is indeed counted as a clone. For example, **Fig. 9** shows how this information is kept in the hole. The hole # in **Fig. 4(b)** has its program points, l_5 and l_{10}, and its hole masses, 1 and 3, representing the sizes of two sub-trees `6` and `*(b,1)`, respectively.

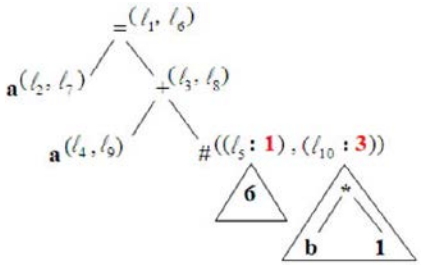


Fig. 9. Example: hole mass in duplicate pattern

Each pattern cluster possesses the following information: the number of tokens excluding holes, the number of holes, a list of hole masses for each hole, and a label list representing program points for each node.

Let us now analyze the asymptotic time complexity of our main algorithm in **Fig. 6**. Let n_1 be the number of nodes in tree t_1, and n_2 be the number of nodes in tree t_2. Then the number of all subtree pairs to be compared is $n_1 \times n_2$ that represents the number of comparisons made at line 24 in **Fig. 6**. Function *anti-unify* is invoked only when function symbols in the root node of t_1' and t_2' are identical. In that case, at least once or at most $min(m_1,m_2)$ comparisons are additionally made where m_1 is the number of nodes in tree t_1' and m_2 is the number of nodes in tree t_2'. Note here that m_1 and m_2 are always smaller than n_1 and n_2. Then in lines 27 and 28 in **Fig. 6**, function *clustering* is invoked once for each function symbol. Letting k be the size of the input pattern set, the condition in line 13 in **Fig. 8** is executed k^2 times in the worst case when input patterns are all in different shape.

4. Evaluation

The algorithm presented in the previous section is implemented in Objective Caml 3.09 and Python 2.5.1, and named CCR, the acronym for Code Clone Ransacker. CCR is basically able to handle the AST representation of programs in any programming language. However, we choose four open-source Java applications listed in **Table 1** for the evaluation of CCR. They are the same applications used by Bellon-Koschke-Krinke-Merlo[12] and Roy-Cordy[13] for their experiments. Joust 0.8[2], a Java parser written in Objective Caml, is modified to parse Java sources and produce abstract syntax trees.

Table 1. Open-source Java Applications used in Experiment

applications	size(MB)	files	lines
netbeans-javadoc	0.69	97	14,301
eclipse-ant	1.35	149	29,880
eclipse-jdtcore	6.37	687	135,675
j2sdk1.4.0-javax-swing	8.32	533	202,943

[2] `http://www.cs.cmu.edu/~ecc/joust.tar.gz`

4.1 Experiment Set-up

A pattern cluster might represent a set of exact clones (type 1) if there is no hole. Otherwise, it might represent a set of clones of type 2 or 3[3]. However, not all pattern clusters represent useful code clones. Some pattern clusters with any of the following properties might not represent useful code clones:

- The number of nodes in a pattern cluster is too small.
- The number of holes in a pattern cluster is too high.
- The hole mass of some hole in a pattern cluster is too big.

Hence, the following three parameters are given to the tool in order to throw away pattern clusters that are less likely to be useful clones.

- `MinNode`: the minimum number of nodes allowed in a pattern cluster.
- `MaxHole`: the maximum number of holes allowed in a pattern cluster.
- `HoleMassLimit`: the largest hole mass allowed in a hole.

That is, CCR ignores pattern clusters with the number of nodes less than `MinNode` as well as those with the number of holes greater than `MaxHole`. CCR also throws away pattern clusters where there is at least one hole whose mass is greater than `HoleMassLimit`. CCR's default parameter values are set to: `MinNode` = 20, `MaxHole` = 15, `HoleMassLimit` = 5. A pattern cluster with 20 nodes corresponds to, on the average, about two to three lines of code. This parameter value equals to the number employed by Evans and Frazer [10]. If the `MaxHole` value is set to 0, CCR only finds exact clones. The `MaxHole` value should be increased if one wants to find clones of type 2 and 3, and is set to 15 that should be big enough for a default value. `HoleMassLimit` has to be set to at least greater than or equal to 5 because the value less than 5 turns out to be too small to be useful.

The experiment is carried out with 3 more sets of parameter values in addition to the default values, while the value of `MinNode` is fixed to 20 all the time. For each of four combinations of parameter values, the number of clustered patterns found by CCR for each application is listed in **Table 2**.

Table 2. The Number of Pattern Clusters Found by CCR

parameter value combinations		no.1	no.2	no.3	no.4
parameters	`MaxHole`	**15**	**15**	**25**	**25**
	`HoleMassLimit`	**5**	**10**	**10**	**25**
netbeans-javadoc		139	151	165	209
eclipse-ant		152	176	211	341
eclipse-jdtcore		1913	2074	2681	3846
j2sdk1.4.0-javax-swing		1061	1380	2092	3452

The experiment has been done on MacPro at Xeon 2*2.8 Quad Core processor and 8GB RAM running Mac OX X Server 10.5.8. **Table 3** shows the execution time of each run for each application.

[3] Type 1, 2, and 3 are according to Koschke's categorization [14].

Table 3. Execution Time

parameter value combinations		no.1	no.2	no.3	no.4
parameters	MaxHole	15	15	25	25
	HoleMassLimit	5	10	10	25
netbeans-javadoc		0h 01m 47s	0h 02m 03s	0h 02m 17s	0h 02m 43s
eclipse-ant		0h 04m 58s	0h 04m 58s	0h 05m 05s	0h 05m 59s
eclipse-jdtcore		2h 46m 28s	2h 50m 47s	3h 14m 30s	3h 47m 36s
j2sdk1.4.0-javax-swing		3h 57m 30s	4h 00m 11s	4h 01m 05s	4h 12m 23s

4.2 Precision

Precision is defined as the ratio of actual clones to the candidates identified by the tool. Each pattern cluster contains line and column numbers in source code where the pattern originates. Hence, the collection of marked blocks corresponding to a pattern cluster in source code can be examined to see if the pattern cluster can be classified as an actual clone. A good portion of output is closely examined with naked eyes in order to identify actual clones. **Table 4** shows the number and ratio of pattern clusters classified as false positives. The enormous number of pattern clusters detected for eclipse-jdtcore and j2sdk1.4.0-javax-swing in the cases of combinations of no.3 and no.4 prevents us from examining the results. All in all, false positives tend to be increased with parameter values increased.

Table 4. The Number of False-positives

parameter value combinations		no.1	no.2	no.3	no.4
parameters	MaxHole	15	15	25	25
	HoleMassLimit	5	10	10	25
netbeans-javadoc		5(3.6%)	8(5.3%)	17(10.4%)	47(22.8%)
eclipse-ant		4(2.6%)	14(8.0%)	35(16.7%)	153(44.8%)
eclipse-jdtcore		34(1.8%)	136(6.6%)	-	-
j2sdk1.4.0-javax-swing		41(3.9%)	102(7.4%)	-	-

We are able to categorize the types of false positives according to the classification of Roy, Cordy and Koschke[14] as follows:

- Category 1: Successive very simple method declarations
- Category 2: Successive method with similar program structure except identifier
- Category 3: Successive try-catch statements, case entries, and if-statements
- Category 4: Successive variable declarations

The number of false positives for each category is shown in **Table 5**.

Table 5. The Number of False-positives for Each Category

parameter value combinations		no.1				no.2				no.3				no.4			
parameters	MaxHole	15				15				25				25			
	HoleMassLimit	5				10				10				25			
Category		1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
netbeans-javadoc		4	1	0	0	4	2	2	0	4	8	2	3	4	33	7	3
eclipse-ant		0	2	2	0	9	12	2	0	9	20	2	4	9	105	31	8
eclipse-jdtcore		2	27	3	2	2	112	20	2	-	-	-	-	-	-	-	-
j2sdk1.4.0-javax-swing		3	25	1	12	3	60	19	2	-	-	-	-	-	-	-	-

The most frequent false positives are in category 2. As shown in **Fig. 10**, false positives in this category share the following common pattern: the program structure is very similar, but types and identifiers are all different.

netbeans-javadoc/DocFSoffsetEditor.java (lines 50-57)

```
50  public void addPropertyChangeListener
                    (PropertyChangeListener l) {
51      super.addPropertyChangeListener(l);
52      supp.addPropertyChangeListener (l); }
53  public void removePropertyChangeListener
                    (PropertyChangeListener l) {
54      super.removePropertyChangeListener(l);
55      supp.removePropertyChangeListener (l); }
56  public boolean supportsCustomEditor () {
57      return true; }
```

netbeans-javadoc/RootDocImplWrapper.java (lines 152-153)

```
152  public void printNotice(String msg) {
153    out.println ( msg );
154    out.flush();
155    }//Res.printNotice(msg);
156  public void printNotice
                    (SourcePosition pos, String msg) {
157    out.println ( msg );
158    out.flush();
159    }//Res.printNotice(msg);
160  public SourcePosition position() {
161    return null; } }
```

Fig. 10. False-positive Case: Successive Methods with Similar Program Structure Except Identifier

A typical false positive of category 1 is shown in **Fig. 11**. The example is a simple list of abstract method declarations that has similar structure, which cannot be considered as clones. False positives of categories 3 and 4 show the similar patterns to category 1.

netbeans-javadoc/ExternalJavadocType.java (lines 36-51)

```
36  public abstract int getMaxmemory ();
37  /** Setter max memory
38  */
39  public abstract void setMaxmemory (int s);
40  /** Getter for recursive
41  */
42  public abstract boolean getRecursive ();
43  /** Setter for recursive
44  */
45  public abstract void setRecursive (boolean rec);
46  /** Getter for external executor for Javadoc
47  */
48  public abstract ServiceType getExternalExecutorEngine();
49  /** Setter for external executor for Javadoc
50  */
51  public abstract void setExternalExecutorEngine
                                    (ServiceType executor);
.....................................................
```

netbeans-javadoc/JavadocType.java (lines 52-63)

```
52  public abstract long getMembers();
53  /** Setter for members
54  */
55  public abstract void setMembers( long l );
56  /** Getter for path to overview file.
57  */
58  public abstract File getOverview ();
59  /** Setter for path to overview file.
60  */
61  public abstract void setOverview (File s);
62  /** Getter for bootclasspath
63  */
64  public abstract String getBootclasspath ();
65  /** Setter for bootclasspath
66  */
67  public abstract void setBootclasspath (String s);
```

Fig. 11. False-positive Case: Successive Very Simple Method Declarations

4.3 Recall

The Bellon's reference corpus [12] is used in evaluating CCR's Recall[4]. The numbers of clone pairs collected by Bellon for selected Java applications are listed in **Table 6**. That is, for each clone pair in the reference corpus, patterns detected by CCR are examined to see if any of them has good-value > 0.7. The examination is straightforward since the exact locations of patterns and clone pairs in the corpus are both available. For two huge reference corpora, a subset (5%) was carefully chosen in such a way that clones of type 1, 2 and 3 are proportionally distributed. The number of reference clone pairs actually used for evaluation is listed in **Table 6** for each clone type.

Table 6. The Number of Clone Pairs in the Bellon's Reference Corpus

applications	the Bellon's reference corpus (# of clone pairs)				selected for experiment			
	type1	type2	type3	total	type1	type2	type3	total
netbeans-javadoc	6 (11%)	32 (58%)	17 (31%)	55	6 (11%)	32 (58%)	17 (31%)	55
eclipse-ant	4 (13%)	24 (80%)	2 (7%)	30	4 (13%)	24 (80%)	2 (7%)	30
eclipse-jdtcore	120 (9%)	866 (64%)	359 (27%)	1345	6 (9%)	43 (64%)	18 (27%)	67
j2sdk1.4.0-javax-swing	145 (18%)	595 (77%)	37(5%)	777	7 (18%)	29 (77%)	2 (5%)	38
total	275 (12.5%)	1517 (68.7%)	415 (18.8%)	2207	23 (12.1%)	128 (67.4%)	39 (20.5%)	190

[4] Details of the experimental data can be found at `http://plasse.hanyang.ac.kr/ccr`

The goal is to see how many clone pairs CCR finds among the reference corpus. The result of experiment is shown in **Table 7**.

Table 7. Recall of CCR

applications	the number of clone pairs with good-value > 0.7														
	reference corpus			combination no.1			combination no.2			combination no.3			combination no.4		
	type1	type2	type3	type1	type2	type3	type1	type2	type3	type1	type2	type3	type1	type2	type3
netbeans-javadoc	6	32	17	6	28	2	6	29	2	6	32	2	6	32	6
eclipse-ant	4	24	2	4	22	2	4	22	2	4	24	2	4	24	2
eclipse-jdtcore	6	43	18	6	43	13	6	43	13	6	43	13	6	43	13
j2sdk1.4.0-javax-swing	7	29	2	7	29	0	7	29	1	7	29	1	7	29	2
total	23	128	39	23	122	17	23	123	18	23	128	18	23	128	21
	190			162			164			169			172		
recall(%)	-	-	-	100	95.3	43.6	100	96.1	46.2	100	100	46.2	100	100	53.8
	-			85.3			86.3			88.9			90.5		

The experiment started with default parameter values. With default parameter values (combination no.1), CCR finds all clone pairs of type 1, but missed six of type 2 and twenty-two of type 3. Hence, we can see that the default parameter values are good enough for detecting clones of type 1. For parameter combination no.2, with only the `HoleMassLimit` value increased to 10, CCR finds one more clone pair of type 2 (Ref. No. 520) – a hole corresponding to a big expression – and another pair of type 3 (Ref. No. 7223) – a hole corresponding to a complete assignment statement. For parameter combination no.3, with the `MaxHole` value additionally increased to 25, CCR finds the rest of clone pairs of type 2 (Ref. No.'s 502, 5725, 886, 2216, and 6276), which implies that the parameter values, `MaxHole` = 25 and `HoleMassLimit` = 10 , might be good enough for detecting clones of type 1 and 2. But no more clone pairs of type 3 was found. For parameter combination no.4, with the `HoleMassLimit` value further increased to 25, CCR is able to find three more clone pairs of type 3 (Ref. No.'s 580, 588, and 7298), still missing many. Among the newly found clone pairs, the first two clone pairs have a hole corresponding to two statements of two lines, and the other clone pair has a hole corresponding to if-statement of two lines.

Up to this point, CCR has missed 18 clones pairs in the selected reference corpus. Among them, one clone pair, Bellon's Ref. No. 2414 (type 3), would have been detected if the HoleMassLimit value were set to a bigger number. This particular clone pair creates a hole mass greater than 25 in the middle, and thus is not selected. The other seventeen clone pairs are all classified as non-contiguous (gapped) clones, and would not be detected even if the parameter values are increased further.

Compared to a similar tree-pattern-based tool, Evans-Fraser-Ma's Asta [8], CCR finds far more clone pairs in the Bellon's reference corpus and thus shows far better recall. **Table 8** shows the comparison between Asta's recall as in their paper and CCR with parameter combination no.1. The difference might be partially due to the fact that CCR uses the top-down approach (as opposed to Asta's bottom-up one) to collect identical tree patterns, which allows overlapped clones to be detected, while Asta might miss them.

Table 8. Comparison of OK-found and Good-found between CCR and Asta (OK-value > 0.7 and Good-value > 0.7)

tools		CCR								Asta	
applications	Bellon's Ref.	ok-value				good-value				ok-value	good-value
		type1	type2	type3	total	type1	type2	type3	total		
netbeans-javadoc	55	6	28	12	46 (83.6%)	6	28	2	36 (65.5%)	20 (36.7%)	7 (12.7%)
eclipse-ant	30	4	22	2	28 (93.3%)	4	22	2	28 (93.3%)	13 (43.4%)	7 (23.3%)
eclipse-jdtcore	67	6	43	18	67 (100.0%)	6	43	13	62 (92.5%)	_ (37.4%)	_ (18.1%)
j2sdk1.4.0-javax-swing	38	7	29	0	36 (94.7%)	7	29	0	36 (94.7%)	_ (58.4%)	_ (33.2%)

Bulychev-Minea's CloneDigger claims to have about 46.7% recall on the average for the same set of applications [15].

5. Towards Better Recall

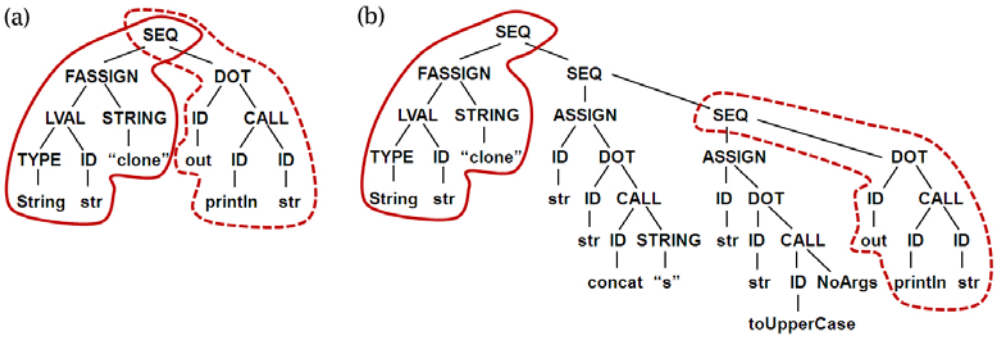
Every exact clone (type 1) has a corresponding 'no-hole' pattern, and every clone of type 2 has a corresponding pattern when the `MaxHole` is set to a large enough value. On the other hand, a clone of type 3 might not have a corresponding pattern when it is either non-contiguous or intertwined. Seventeen out of eighteen false negatives found in our experiment are non-contiguous clones. In this section, we propose how to effectively find non-contiguous clones and intertwined clones.

In a program tree, two patterns are called *neighbors* if they share a common node, are adjacent, or are not too far apart in a predictive manner. Suppose that given two program trees t_1 and t_2, two duplicate patterns p_1 and p_2 are all found distinctively in both t_1 and t_2, and the continuous program fragment corresponding to p_i in t_j is c_{ij}. If p_1 and p_2 are neighbors, the continuous program fragment comprising both c_{1k} and c_{2k} may be combined to become a program fragment c_k for each $k \in \{1, 2\}$. Then c_1 and c_2 are also clones.

One typical type of neighbor patterns is that two patterns are on the same spine in a program tree. Consider the following two programs[5]:

```
(a)                          (b)
String str = "clone";        String str = "clone";
out.println(str);            str = str.concat("s");
                             str = str.toUpperCase();
                             out.println(str);
```

The corresponding program trees showing two patterns are:



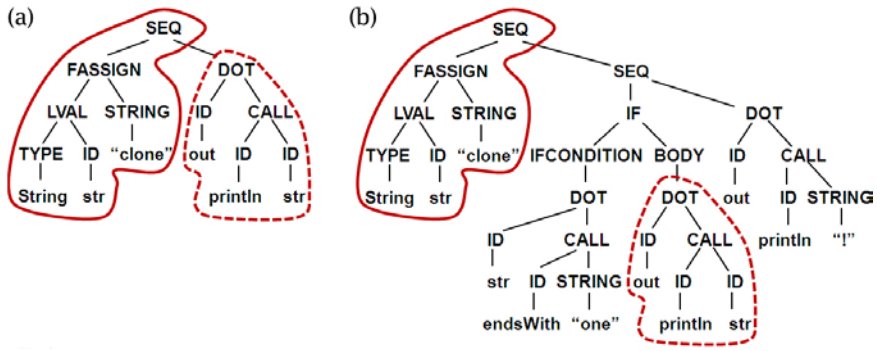
[5] These two programs are too short to be a useful clone, but they are selected on purpose just to make the presentation simple.

These two patterns can be regarded as neighbors because they are not contiguous but on the same spine of `SEQ` nodes. This type of pattern neighbor is found in sequentially noncontiguous clones. In fact, fourteen clone pairs missed by CCR in our experiment, - Bellon's Ref. No.'s 581, 589, 5849, 5866, 6061, 6248, 6444, 6615, 6616, 7382, 7392, 7396, 7408, and 7429 - can be classified as this type.

Two patterns, where one resides down below to the spine from another, could be considered as neighbors if they are separated in a predictive manner. For example, consider the following two programs:

```
(a)                        (b)
String str = "clone";      String str = "clone";
out.println(str);          if (str.endsWith("clone"))
                               out.println(str);
                           out.println("!");
```

The corresponding program trees showing two patterns are:



These two patterns can be regarded as neighbors because they are not contiguous but nested inside an `IF` statement tree. This type of pattern neighbor is found in non-contiguous clones in which a pattern resides inside a block tree. Two clone pairs missed by the tool in our experiment, Bellon's Ref. No.'s 6089 and 590, can be classified as this type.

A leftover false-negative clone, Ref. No. 579, contains both types discussed above. No intertwined clone is observed in our experiment, but can also be detected by applying the same idea of combining neighbor patterns.

All these findings have been performed manually with pencil and paper. We are currently developing a method of identifying neighbor patterns algorithmically, hoping to detect non-contiguous clones and intertwined clones mechanically.

6. Related Works

Techniques for automatic clone detection are usually classified into seven categories as in **Table 9**. Each technique has pros and cons. Tree-based technique is known to be more accurate than string-based and token-based ones due to its use of structural information. PDG-based and metric-based techniques are very effective in some cases, but may result in false positives because different code fragments may have the same graph and metric, respectively. String-based technique is sensitive in white space insertion and identifier change, and thus may have many unwanted false negatives. Token-based technique is less sensitive than string-based one because it ignores white spaces and comments, but still has room for false negatives due to its lack of grammatical information. Tree-based technique takes

grammatical structure into account and has potential for better results.

Table 9. Taxonomy of clone detection tools

Classification	Comparison unit	Available tools
String-based	Character	diff in UNIX
Token-based	Token	Baker's Dup [1], CP-Miner [4], CCFinder [3], Iclone [15]
Tree-based	AST (abstract syntax tree)	Yang's [7], Baxter et al.'s CloneDR® [6], Jiang et al's DECKARD [5], Evans et al.'s Asta [8], Bulychev et al.'s CloneDigger [16], Lee and Doh's [9]
Metric-based	Metric values	Mayrand et al. [17], Patenaude et al. [18]
PDG-based	Program dependency graph	PDG-DUP [19], GPLAG [20]
Hybrid	Some combination of the above	Leitao[21] (mixes syntactic and semantic approaches), Li et al. [22, 23] (mixes syntactic and textual approaches)

Yang [7] proposed an approximation algorithm to find the syntactic differences between two versions of the same programs through the tree editing distance. Baxter et al.'s CloneDR®[6, 29] is the representative work among tree-based techniques. CloneDR® generates an annotated parse tree and compares its sub-trees by characterization metrics based on a hash function through tree matching [13]. Jiang et al.'s DECKARD[5] is a tool for detecting code clones and copy-and-paste bugs on large code bases. Characteristic vectors of the binary tree representation of a program are computed to approximate the structural information in the Euclidean space. Its clever characterization of a tree makes it scalable, but may result in false positives due to the loss of some structural information. Milea et al.[30] also makes use of characteristic-vectors to encode code changes for refactorings and detects both refactoring opportunities and historical refactorings in large code bases. Wahler et al. [31] uses the frequent item-set technique for clone detection to detect exact and parameterized clones in the XML representation of AST. Koschke et al. [32] and Tairas et al. [33] introduced linear-time clone-detection algorithms based on suffix tree. Corazza et al. [34] proposes a Tree Kernel to compare abstract syntax trees that are a class of functions for computing similarity among information arranged.

Evans et al.'s Asta [8] proposes to compare ASTs to find clones in the form of tree patters with gaps, which are essentially identical to tree-patterns in CCR. One notable difference is that Asta uses bottom-up approach of dynamic programming, while CCR uses top-down approach by maintaining a memo table to avoid repeated comparison. Asta may fail to report overlapping clones, while CCR does not due to its adoption of a top-down approach. Bulychev et al.'s CloneDigger [16] proposes a method of detecting duplicate codes using anti-unification. The fundamental difference form CCR is the unit of a tree pattern. CloneDigger's clone unit is restricted to be a statement or a statement sequence, but CCR has no restriction. Li et al. proposes to compare pre-processed suffix trees and examine clone candidates using anti-unification for reducing false positives [22,23].

7. Conclusion and Future Work

We designed and implemented CCR, a tree-pattern-based code clone detector. We carried out experiments for standard open-source Java applications, showing that CCR finds clones of type 1 and 2 well, and has very high precision and recall. We also discussed how noncontiguous clones (and even intertwined clones) missed by CCR can be detected using the concept of neighbor patterns.

We are currently exploring how to algorithmically find neighbor patterns, which will eventually make it possible to detect more complicated clones such as non-contiguous clones and intertwined clones automatically. What will follow is to build a reliable and compact reference corpus that can be used to evaluate clone detection tools.

References

[1] Brenda S. Baker, "On finding duplication and near-duplication in large software systems," in *Proc. of the Working Conf. on Reverse Engineering*, pp.86-95, July 14-16, 1995. Article (CrossRef Link)

[2] Elizabeth Burd and John Bailey, "Evaluating clone detection tools for use during preventive maintenance," in *Proc. of the IEEE Int. Workshop on Source Code Analysis and Manipulation*, pp.36-43, October 1, 2002. Article (CrossRef Link)

[3] Toshihiro Kamiya, Shinji Kusumoto and Katsuro Inoue, "CCFinder: a multi-linguistic token based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp.654-670, July, 2002. Article (CrossRef Link)

[4] Zhenmin Li, Shan Lu, Suvda Myagmar and Yuanyuan Zhou, "CP-Miner: finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp.176-192, March, 2006. Article (CrossRef Link)

[5] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su and Stephane Glondu, "DECKARD: scalable and accurate tree-based detection of code clones," in *Proc. of Int. Conf. of Software Engineering*, pp.96-105, May 20-26, 2007. Article (CrossRef Link)

[6] Ira D. Baxter, Andrew Yahin, Leonard Moura, Marcelo Sant'Anna and Lorraine Bier, "Clone detection using abstract syntax trees," in *Proc. of Int. Conf. on Software Maintenance*, pp.368-377, March 16-19, 1998. Article (CrossRef Link)

[7] Wuu Yang, "Identifying syntactic differences between two programs," *Software - Practice and Experience*, vol. 21, no. 7, pp.739-755, June, 1991. Article (CrossRef Link)

[8] William S. Evans, Christopher W. Fraser and Fei Ma, "Clone detection via structural abstraction," *Software Quality Journal*, vol. 17, no. 4, pp.309-330, December, 2009. Article (CrossRef Link)

[9] Hyo-Sub Lee and Kyung-Goo Doh, "Tree-pattern-based duplicate code detection," in *Proc. of ACM Int. Workshop on Data-Intensive Software Management and Mining*, pp.7-12, November 6, 2009. Article (CrossRef Link)

[10] Gordon D. Plotkin, "A note on inductive generalization," *Machine Intelligence*, vol. 5, pp.153-163, 1970. Article (CrossRef Link)

[11] John C. Reynolds, "Transformational systems and the algebraic structure of atomic formulas," *Machine Intelligence*, vol. 5, pp.135-151, 1970. Article (CrossRef Link)

[12] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jns Krinke and Ettore Merlo, "Comparison and evaluation of clone detection tools," *IEEE Tranactions on Software Engineering*, vol. 33, no. 9, pp.577-591, September, 2007. Article (CrossRef Link)

[13] Chanchal K. Roy and James R. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR 2007-541*, 115, September 26, 2007. Article (CrossRef Link)

[14] Chanchal K. Roy, James R. Cordy and Rainer Koschke, "Comparison and evaluation of code clone detection techniques and tools: a qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp.470-495, 2009. Article (CrossRef Link)

[15] Jan Harder and Nils Göde, "Efficiently handling clone data: RCF and Cyclone," in *Proc. of the 5th International Workshop on Software Clones*, pp.81-82, ACM 2011. Article (CrossRef Link)

[16] Peter Bulychev and Marius Minea, "An evaluation of duplicate code detection using anti-unification," in *Proc. of Int. Workshop on Software Clones*, March 24-27, 2009. Article (CrossRef Link)
[17] Jean Mayrand, Claude Leblanc and Ettore Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proc. of Int. Conf. on Software Maintenance*, pp.244-253, November 4-8, 1996. Article (CrossRef Link)
[18] Jean-Francois Patenaude, Ettore Merlo, Michel Dagenais and Bruno Lague, "Extending software quality assessment techniques to Java systems," in *Proc. of Int. Workshop on Program Comprehension*, pp.49-56, May 5-7, 1999. Article (CrossRef Link)
[19] Raghavan Komondoor and Susan Horwitz, "Using slicing to identify duplication in source code," in *Proc. of Int. Symp. on Static Analysis*, pp.40-56, July 40-56, 2001. Article (CrossRef Link)
[20] Chao Liu, Chen Chen, Jiawei Han and Philip S. Yu, "GPLAG: detection of software plagiarism by program by program dependence graph analysis," in *Proc. of ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pp.872-881, August 20-23, 2006. Article (CrossRef Link)
[21] Antonio M. Leitao, "Detection of redundant code using R2D2," *Software Quality Journal*, vol. 12, no. 4, pp.361-382, December 2004. Article (CrossRef Link)
[22] Huiqing Li and Simon Thompson, "Similar code detection and elimination for Erlang programs," in *Proc. of Int. Symp. on Practical Aspects of Declarative Languages*, pp.104-118, January 18-19, 2010. Article (CrossRef Link)
[23] Huiqing Li and Simon Thompson, "Incremental code detection and elimination for Erlang programs," in *Proc. of Int. Conf. on Fundamental Approaches to Software Engineering*, pp.356-370, March 26 – April 3, 2011. Article (CrossRef Link)
[24] Brenda S. Baker and Udi Manber, "Deducing similarities in Java sources from bytecodes," in *Proc. of USENIX Annual Technical Conference*, pp.15-18, June 15-19, 1998. Article (CrossRef Link)
[25] Ian J. Davis and Michael W. Godfrey, "From whence it came: detecting source code clones by analyzing assembler," in *Proc. of Working Conference on Reverse Engineering*, pp.242-246, October 13-16, 2010. Article (CrossRef Link)
[26] Andrew Saebjoernsen, Jeremiah Wilcock, Thomas Panas, Daniel Quinlan and Zhendong Su, "Detecting code clones in binary executables," in *Proc. of Int. Symp. on Software Testing and Analysis*, pp.117-128, July 19-23, 2009. Article (CrossRef Link)
[27] Antonella Santone, "Clone detection through process algebras and Java bytecode," in *Proc. of Int. Workshop on Software Clones*, pp.73-74, May 23, 2011. Article (CrossRef Link)
[28] Heejung Kim, Yungbum Jung, Sunghun Kim and Kwangkeun Yi, "MeCC: memory comparison-based clone detector," in *Proc. of Int. Conf. of Software Engineering*, pp.301-310, May 21-28, 2011. Article (CrossRef Link)
[29] Ira D. Baxter, Christopher Pidgeon and Michael Mehlich, "DMS: Program transformations for practical scalable software evolution," in *Proc. of Int. Conf. on Software Engineering*, pp.625-634, May 23-28, 2004. Article (CrossRef Link)
[30] Narcisa A. Milea, Lingxiao Jiang, and Siau-Cheng Khoo, "Vector abstraction and concretization for scalable detection of refactorings," in *Proc. of the ACM SIGSOFT Int. Symp. on the Foundations of Software Engineering*, November 16-21, pp.86-97, 2014. Article (CrossRef Link)
[31] Vera Wahler, Dietmar Seipel, Jurgen Wolff v. Gudenberg and Gregor Fischer, "Clone detection in source code by frequent itemset techniques," in *Proc. of IEEE Int. Workshop on Source Code Analysis and Manipulation*, pp.128-135, September 15-16, 2004. Article (CrossRef Link)
[32] Rainer Koschke, Raimar Falke and Pierre Frenzel, "Clone detection using abstract syntax suffix trees," in *Proc. of Working Conf. on Reverse Engineering*, pp.253-262, October 23-27, 2006. Article (CrossRef Link)
[33] Robert Tairas and Jeff Gray, "Phoenix-based clone detection using suffix trees," in *Proc. of ACM Annual Southeast Regional Conference*, pp.679-684, March 10-12, 2006. Article (CrossRef Link)
[34] Anna Corazza, Sergio Di Martino, Valerio Maggio and Giuseppe Scanniello, "A tree kernel based approach for clone detection," in Proc. of Int. Conf. on Software Maintenance, pp.1-5, September 12-18, 2010. Article (CrossRef Link)

Hyo-Sub Lee received the B.S. degree in computer science from Daejin University in 1997, the M.S. and Ph.D. degrees in computer science from Hanyang University in 2000 and 2013, respectively. She is now an industry professor at the Department of Computer Science at Hanyang University. Her primary research interests are software engineering and program analysis.

Myung-Ryul Choi received the B.S. degree in electronic engineering from Hanyang University in 1983 and the M.S. and Ph. D. degrees in electrical engineering from Michigan State University in 1985 and 1991, respectively. From 1991 to 1992, he was with the Korea Institute of Industrial Technology where he was an assistant professor. In 1992, he joined the Department of Electrical Engineering and Computer Science at Hanyang University, where he is now a professor. His current research interests include digital/analog VLSI, 2D/3D FPD controller, RFID, ITS, and smart card applications.

Kyung-Goo Doh (corresponding author) received the B.S. degree in industrial engineering from Hanyang University in 1980, the M.S. degree in computer science from Iowa State University in 1987, and the Ph.D. degree in computer science from Kansas State University in 1992. From 1993 to 1995, he was with the University of Aizu as an assistant professor. He then joined the Department of Computer Science at Hanyang University ERICA, where now is a professor. His primary research interests are programming languages, program analysis, software engineering and software security.