A Simplified Approach to Parameterized Clone Detection Using Abstract Syntax Tree

Chung Yung, Che-Wei Wu

National Dong Hwa University,
Department of Computer Science and Information Engineering,
Hualien, Taiwan, R.O.C.

Abstract

This paper describes a new approach of parameterized clone detection using abstract syntax tree. Since a considerable fraction of the source code in large-scale computer programs is duplicate code, called as clones, replacing the duplicate code by procedure calls serves as an effective way for reducing object code size. Various techniques are proposed for finding clones, but not much about their use with procedure abstraction is reported. We propose a new approach of parameterized clone detection using abstract syntax tree so that a straightforward transformation on the detected clones into procedure calls is possible.

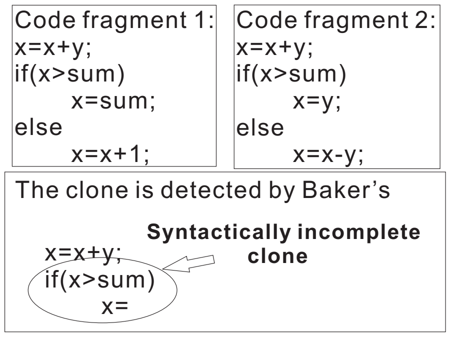
Key words: Clone detection, redunant code, procedure abstraction, abstract syntax tree, parameterized clone.

1 Introduction

In this paper, we propose a new approach to *parameterized clone detection* so that the clones found may be replaced by procedure calls. It is reported that it may reduce the object code size up to 15% when applying procedure abstraction on the source code[24]. The main idea of procedure abstraction is to abstract the identical parts in a program into a single procedure, which calls for a clone detection technique that is good for the purpose of procedure abstraction.

Email addresses: `yung@mail.ndhu.edu.tw` (Chung Yung), `m9521038@em95.ndhu.edu.tw` (Che-Wei Wu).

Technical Report CTA-TR-200801



```
Code fragment 1:
x=x+y;
if(x>sum)
      x=sum;
else
      x=x+1;
```

```
Code fragment 2:
x=x+y;
if(x>sum)
      x=y;
else
      x=x-y;
```

The clone is detected by Baker's

```
x=x+y;
if(x>sum)
      x=
```

Fig. 1. A syntactically incomplete clone.

A recent research suggests that large software systems typically contain 10-25% redundant code, or clones[19]. The redundancy is caused by reusing code through copy-and-paste or by programming the structural similarities code accidentally. If there are many clones in a program, the redundancy makes the code unstructured and large. As a result, more memory is required for the object code, and more cost is spent for maintaining the code. This shows the importance of detecting and refactoring the code clones.

On the other hand, detection and refactoring of clones by procedure calls promises decreased program maintenance cost corresponding to the reduction in code size. There are many clone detection techniques proposed from the aspect of software engineering, but the focus of these approaches does not aim at procedural abstraction. The proposed techniques usually give the constrains on the clones found without taking the language constructs into consideration so that some detected clones are syntactically incomplete. For the example in Figure 1, the detected clone of the two code fragments is syntactically incomplete. The new approach we propose is facilitated for detecting the parameterized clones that are syntactically complete.

According to Rysselberghe and Demeyer[21], the clone detection techniques can be divided into three categories: the *string-based* techniques[6,10], the *token-based* techniques[3,13], and the *abstract-syntax-tree(AST)-based* techniques[5,11,15,18]. The string-based clone detection divides the program into a number of strings such as lines, and the whole lines are compared with each other textually[6]. The token-based clone detection transforms the program into a token stream and then search the similar token sequence using suffix tree. The AST-based clone detection finds similar subtrees after building a parse tree.

(1)	(2)
$X = A + B$	$X = P + Q$
$Z = A - B$	$Z = P - Q$
(3)	(4)
$X = P + Q$	$X = P + 1$
$Z = A - B$	$Z = P - 1$
(5)	(6)
$X = P * Q$	$X = 5$
$Z = P / Q$	$Z = 10$

Fig. 2. The motivation example.

We are interested in those fragments in the source code that can be transformed into procedure calls. As an example, consider the six different code fragments in Figure 2, most clone detection techniques find that the code fragment (2) is a clone of (1), while code fragments (3), (4), (5), and (6) are not considered as clones of (1). However, it is clear that all six code fragments may be abstracted into a single procedure.

This paper is orgainzed as following. The next section is a brief description and comparsion on the related works. Two of the popular clone detection techniques are presented in particular; namely, the Baxter's AST-based approach and the Baker's token-based approach. In section 3, we give the definition of parameterized clones and the terminology used in this paper. In section 4 we propose a new approach of clone detection. Section 5 presents an example of applying our technique in detecting clones. At last is a brief conclusion.

2 Related Works

There are many different approaches proposed to find clones in the source code [6,10,3,13,5,11,15,18], but the definition of a clone in different approaches is varied. The approach proposed in this pape an AST-based technique with the concept of using a suffix tree adaptd from the token-based techniques. Hence, we introduce an AST-based approach proposed by Baxter et al.[5], and a token-based approach proposed by Baker[3] in this section.

2.1 Baxter's AST-based Approach

A clone detection approach based on abstract syntax trees is proposed by Baxter et al.[5]. The basic idea is comparing each subtree of the abstract syntax tree by computing the similarity. If there are two subtrees whose similarity exceeds a threshold, these subtrees are called clone. The similarity is computed by following formula :

Similarity = 2*S / (2*S+L+R)
where:
S = # of shared nodes
L = # of different nodes in subtree 1
R = # of different nodes in subtree 2

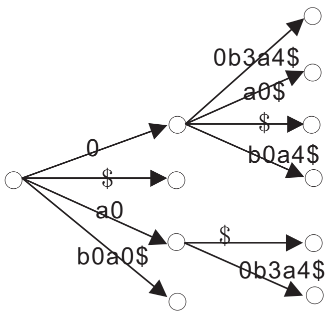
But computing the similarities of all subtree pairs is not efficient, of which the computation complexity is O(N^3), where N is the number of nodes in the AST. In order to solve this problem, Baxter et al. propose using a hash function to hash subtrees into some buckets if the subtree mass exceeds the mass threshold.

The basic algorithm proposed by Baxter et al. is presented in Figure 3(a). The algorithm finds single-subtree clones, but the subtree-sequence clones could not be detected. Baxter et al. build a list structure where each list is associated with a sequence in the program. The hash codes of each subtree element in the associated sequence are stored in the list structure. The Figure 3(b) gives the clone detection algorithm for subtree-sequence clones, which compares each pair of subtree-sequence clones if the length of the clone exceeds the given threshold. This idea is similar to the clone detection of single-subtree clones, which compares all pairs of subtrees. Note that a hash function is used in this algorithm to hash the subtrees and the subtree sequences into buckets.

The clone detection technique proposed by Baxter et al. compares each pair of subtrees or subtree sequences to find the exact and near-miss clones. Because of the tree matching used in their approach, the computation complexity is high.

2.2 Baker's Token-based Approach

The idea of token-based clone detection is proposed by Baker[1,2,3,4]. Baker's approach detects the exact and parameterized matched clones, while two code fragments are called a parameterized match (*p*-match) if there is a one-to-one function that maps the set of parameters in one fragment onto the set of pa-



A p-suffix tree of the p-string $\mathcal{S}$ = axybxay$
Which the encoded substrings are:
a00b3a4$, 00b3a4$, 0b0a4$, b0a0$, 0a0$, a0$, 0$, $

Fig. 3. An example of p-suffix tree of p-string $\mathcal{S}$.

rameters in another fragment.

For any program code, this approach uses the lexical analyzer to generate a token string consisting of one "non-parameter symbol" and zero or more "parameter symbols". A code fragment such as x=x+y is first transformed into a pattern P=P+P and a list x,x,y; then a non-parameter symbol is generated to represent the pattern P=P+P and three parameter symbols are generated to represent x,x,y. In this way, both the parameter candidates and their positions are recorded in the resulting string, which is called a *parameterized string* or a *p-string*.

In order to detect a p-match from p-strings, this approach also needs to encode the parameter symbols as follows. The first occurrence of each parameter symbol is replaced by a 0. Each later occurrence of parameter symbol is replaced by the distance in the string since the previous occurrence of the same symbol. Non-parameter symbols are left unchanged. For example, if a, b, and *$* are the non-parameter symbols, and x and y are the parameter symbols, a p-string *axybxay$* would be encoded as *a00b3a4*.

After the lexical analysis, Baker proposes an algorithm ***dup*** for detecting parameterized match. ***dup*** builds a tree called *parameterized suffix tree(p-suffix tree)* for the p-string. When we construct a p-suffix tree for a p-string $\mathcal{S}$, we need to encode each suffix substring of $\mathcal{S}$. In the above example $\mathcal{S}$, *axybxay$*, $\mathcal{S}$ has 8 suffix substrings; namely, *axybxay$, xybxay$, ybxay$, bxay$, xay$, ay$, y$, $*. A suffix tree is a representation of a string as a trie where every suffix is presented through a path from the root to a leaf. For any two leaves, we have two distinct paths from the root to the leaves. If there is a common prefix in the path, the shared path is a p-match substring of the p-strings. In this case, the substring is a clone. Figure 3 shows the example of constructing suffix trees for $\mathcal{S}$=*axybxay$*, where *a, b,* and *$* are non-parameter symbols and *x, y* are parameter symbols.

Baker's token-based clone detection using suffix tree is linear in space with respect to the token string length, and the linear algorithm to construct suffix trees is proposed by McCreight[17].

3 A Simple Language

Abstract Syntax of $\mathcal{L}_s$ Language

c	$\in$	*Con*	Constant
x	$\in$	*Var*	Variables
t	$\in$	*Typ*	Types
			$t = int \mid char \mid float$
d	$\in$	*Dec*	Declaration
			$d = t\ x$
op	$\in$	*Pf*	Primitive function
			$op = + \mid - \mid * \mid / \mid > \mid \geqq \mid < \mid \leqq \mid == \mid \neq \mid \&\& \mid \Vert$
f	$\in$	*Fv*	Function variables
fd	$\in$	*Fd*	Function definition
			$fd = t\ f_i(x_1,\ldots,x_n) = \{d_1;\ldots d_j;\ \mathrm{s}\}$
e	$\in$	*Exp*	Expression
			$e = c \mid x \mid e_1\ op\ e_2 \mid f_i(x_1,\ldots,x_n)$
s	$\in$	*Stm*	Statements
			$s = s_1;\ldots s_k; \mid x = e \mid \mathrm{if}(e)\ s_1\ \mathrm{else}\ s_2 \mid \mathrm{while}(e)\ s$
pr	$\in$	*Prog*	Program
			$pr = \{fd_1\ldots fd_m\}$

Fig. 4. Abstract Syntax of $\mathcal{L}_s$ Language

In this section, we propose a simple programming language $\mathcal{L}_s$ which simplifies standard $\mathcal{C}$ language but not to lose the essence of programmming language. We would describe our algorithm based on the $\mathcal{L}_s$ language later, and this language could help us decribing simply and clearly. The $\mathcal{L}_s$ language is a complete and workable programming langugage, and it is strongly typed.

Ervery legal expression has a type which is determined by programer or the compiler. The Figure 4 describe the abstract syntax of $\mathcal{L}_s$ language.

4 Parameterized Clone

In this paper, we are interested in the code fragments that can be transformed into procedure calls. We call such code fragments as *parameterized clones*. Here, we give the definition of a parameterized clone. Note that this definition may be slightly different from those used in the other approaches of clone detection. We will use this definition in the rest of this paper.

***Definition*: A parameterized clone class $\mathcal{C}$ of a program $\mathcal{P}$ is a set of code segments $p_1, ..., p_k$ in $\mathcal{P}$ such that when a general form of $p_1, ..., p_k$ is abstracted into a procedure $\mathcal{F}$, and $p_1, ..., p_k$ can be replaced by calls to $\mathcal{F}$; namely, $f_1, \ldots, f_k$, with appropriate parameters. In such a case, we call p_i is a clone instance of the clone class $\mathcal{C}$, where $1 \leq i \leq k$.** $\Box$

For a program $\mathcal{P}$, the clone set Θ is a set of parameterized clones classes $\{\mathcal{C}_1 \ldots \mathcal{C}_n\}$, and a parameterized clone class $\mathcal{C}$ is a set of clone instances $\{p_1 \ldots p_k\}$ each of which can be transformed into a single procedure call.

Code clones are generally known as duplicated code fragments through copy-and-paste in a software system [12]. In convention, the clone detection techniques are designed to compare two code fragments and decide whether they are similar or not by computing the similarities or matching patterns, such as [5,3]. In this paper, we propose that whether two code fragments are similar or not is decided by whether they can be transformed into procedure calls to the same procedure or not.

For the example in the Figure 2, there are 6 code fragments. The clone class detected by Baker is $\{(1), (2)\}$, in which code fragments are p-match; and the clone class detected by Baxter et al. is $\{(1), (2), (3)\}$, in which the nodes of the subtree in the code fragments are the same. Both of them do not consider (4),(5),and (6) as members in the clone class. In the next section, we present a new approach that can detect that (1),(2),(3),(4),(5),and (6) are all members of the parameterized clone class since each of them can be transformed into a single procedural call to the same procedure.

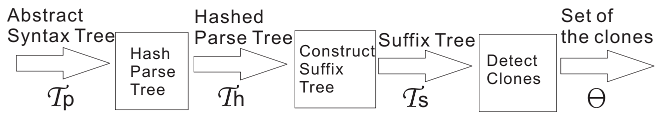


Fig. 5. The framework of our approach.

Note that the clone detected by our approach may include loop statements, selection statements, and calls to first-order functions. The approach that we propose does not consider higher-order functions, but it may be easily extended to include calls to higher-order functions in the clones.

5 Clone Detection

In this section, we propose a new approach for clone detection using abstract syntax tree. Our approach is a framework that can be divided into three steps. First, we hash the parse tree of the program. We use a hash function that hashes the subtree of a statement into a hash key. Second, we construct the suffix tree from the parse tree with hashed keys. For constructing the suffix tree, there are two comparison functions, of which one compares two hashed keys, and the other recursively compares the hashed keys in the subtrees of a compound statement. At last, we compute the clone set of the program from the suffix tree. In this step, two threshold numbers are needed: one for the minimal number of statements in a clone class, and the other for the minimal number of repetitions.

The framework of our approach is shown as Figure 5. We describe each of the steps in our framework in more details.

5.1 Hashing Parse Tree

The first step in our framework of clone detection is to hash the parse tree of a program into a hashed tree. The idea of using hash functions for clone detection is a popular methodology for token-based clone detection [1,2,3,4]. We adapt the hash function to hash the nodes in a parse tree into hash keys.

Our hash function $\mathcal{H}$ hashes the statement s based on the simple language $\mathcal{L}_s$.

Algorithm: Hash Algorithm

$$\mathcal{H}[[s_1;\ldots s_k;]] = \mathcal{S}_c(\mathcal{H}[[s_1]], \mathcal{H}[[s_2;\ldots s_k;]])$$

$$\mathcal{H}[[\text{if}(e)\ s_1 \text{ else } s_2]] = \mathcal{S}_i(\text{e}, \mathcal{H}[[s_1]], \mathcal{H}[[s_2]])$$

$$\mathcal{H}[[\text{while}(e)\ s]] = \mathcal{S}_w(\text{e}, \mathcal{H}[[s_1]])$$

$$\mathcal{H}[[x = e]] = \begin{cases} \mathcal{A}_i & if \quad x.type = int, \\ \mathcal{A}_c & if \quad x.type = char, \\ \mathcal{A}_f & if \quad x.type = float. \end{cases}$$

$$\mathcal{T}[[d]] = \mathcal{T}[[t\ x]] \quad :\equiv \quad x.type = t$$

$$\mathcal{T}[[fd]] = \mathcal{T}[[t\ f_i(e_1,\ldots,e_n)\text{=}\{d_1,\ldots,d_j,s\}]] :\equiv \quad f_i.type = t$$

Fig. 6. The Hash Algorithm.

We denote the hash algorithm $\mathcal{H}$: $\mathcal{S} \rightarrow \mathcal{T}_h$, where $\mathcal{S}$ is the set of $\forall s \in Stm$ in the abstract syntax of $\mathcal{C}_s$ language, and $\mathcal{T}_h$ is the hashed parse tree.

For the simple language $\mathcal{L}_s$ introduced in section 3, the set of hash keys of hashed parse tree is $\{\mathcal{A}_i, \mathcal{A}_c, \mathcal{A}_f, \mathcal{S}_i, \mathcal{S}_w, \$\}$. The hash algorithm is shown in the Figure 6.

The Figure 6 performs the hash algorithm how to hash the statement $s \in Stm$ into a single hash key or the hash keys list. For the statement $s = \{x = e\}$, it would be hashed into a single hash key $\mathcal{A}_i$, $\mathcal{A}_c$, or $\mathcal{A}_f$ according to the type of variable x. For the statement $s = \{if(e_1)\ s_1\ else\ s_2 \mid while(e)\ s\}$, it would be hashed into a single hash key $\mathcal{S}_i$ or $\mathcal{S}_w$ respectively and the hash algorithm still recursively hashes the nested statements $\{s_1,\ s_2\}$ or $\{s\}$. In the hashed parse tree $\mathcal{T}_h$, the children of hash keys $\mathcal{S}_i$ and $\mathcal{S}_w$ are $\{\mathcal{H}[[s_1]],\ \mathcal{H}[[s_2]]\}$ and $\{\mathcal{H}[[s]]\}$ respectively. For the statement $s = \{s_1, \ldots, s_k\}$, it would be hashed into hash keys list for the statements s_1 to s_k and the hash key \$ is added for marking the end of statement s. The example of hashing the parse tree of an example program is shown in Figure 8.

According to above hash algorithm, we develop the compare functions $\mathcal{C}_h$ and $\mathcal{C}_e$ to compare the hash keys of two nodes. The compare functions $\mathcal{C}_h$ and $\mathcal{C}_e$ are shown in Figure 7, and we can see that we recursively call $\mathcal{C}_h$ to compare the subtree of body of two noeds if the hash keys of two nodes both are $\mathcal{S}_w$ or $\mathcal{S}_i$. And we also develop the $\mathcal{C}_e$ function to compare the *condiftion* of each $\mathcal{S}_w$ and each $\mathcal{S}_i$ that is in *while* loop.

Compare Function $\mathcal{C}_h$ and $\mathcal{C}_e$

$$\mathcal{C}_h(h_1,h_2) = \begin{cases} true \;\; if & in\_loop = 0\ \&\\ & h_1 = h_2\ \&\\ & h_1, h_2 \in \{\mathcal{A}_i, \mathcal{A}_c, \mathcal{A}_f\}\\ \mathcal{C}_e(e_1,e_2) \;\; if & in\_loop > 0\ \&\\ & h_1 = h_2\ \&\\ & h_1, h_2 \in \{\mathcal{A}_i, \mathcal{A}_c, \mathcal{A}_f\}\\ \mathcal{C}_h(j_1,j_2)\ \&\ \mathcal{C}_h(k_1,k_2) \;\; if & in\_loop = 0\ \&\\ & h_1 = \mathcal{S}_i(e_1,j_1,k_1)\ \&\\ & h_2 = \mathcal{S}_i(e_2,j_2,k_2)\\ \mathcal{C}_e(e_1,e_2)\ \&\ \mathcal{C}_h(j_1,j_2)\ \&\ \mathcal{C}_h(k_1,k_2) \;\; if & in\_loop > 0\ \&\\ & h_1 = \mathcal{S}_i(e_1,j_1,k_1)\ \&\\ & h_2 = \mathcal{S}_i(e_2,j_2,k_2)\\ \mathcal{C}_e(e_1,e_2)\ \&\ in\_loop+1,\ \mathcal{C}_h(j_1,j_2),\ in\_loop-1 \;\; if & h_1 = \mathcal{S}_w(e_1,j_1)\ \&\\ & h_2 = \mathcal{S}_w(e_2,j_2)\\ \mathcal{C}_h(j_1,j_2)\ \&\ \mathcal{C}_h(k_1,k_2) \;\; if & h_1 = \mathcal{S}_c(j_1,k_1)\ \&\\ & h_2 = \mathcal{S}_c(j_2,k_2)\\ false \;\; otherwise. & \end{cases}$$

$$\mathcal{C}_e(e_1,e_2) = \begin{cases} true \;\; if & e_1 = c\ \&\ e_2 = c\\ true \;\; if & e_1 = x\ \&\ e_2 = x\\ check\_dependence(e_1) \;\; if & e_1 = x\ \&\ e_2 = c\\ check\_dependence(e_2) \;\; if & e_1 = c\ \&\ e_2 = x\\ \mathcal{C}_e(e_3,e_5)\ \&\ \mathcal{C}_e(e_4,e_6) \;\; if & e_1 = e_3\ op_1\ e_4\ \&\\ & e_2 = e_5\ op_2\ e_6\ \&\\ & op_1 = op_2\\ \mathcal{C}_e(e_{11},e_{21})\ \&\ \ldots\ \&\ \mathcal{C}_e(e_{1n},e_{2m}) \;\; if & e_1 = f_1(e_{11},\ldots,e_{1n})\ \&\\ & e_2 = f_2(e_{21},\ldots,e_{2m})\ \&\\ & f_1 = f_2\ \&\ n = m\\ false \;\; otherwise. & \end{cases}$$

Fig. 7. The Compare Function $\mathcal{C}_h$ and $\mathcal{C}_e$.

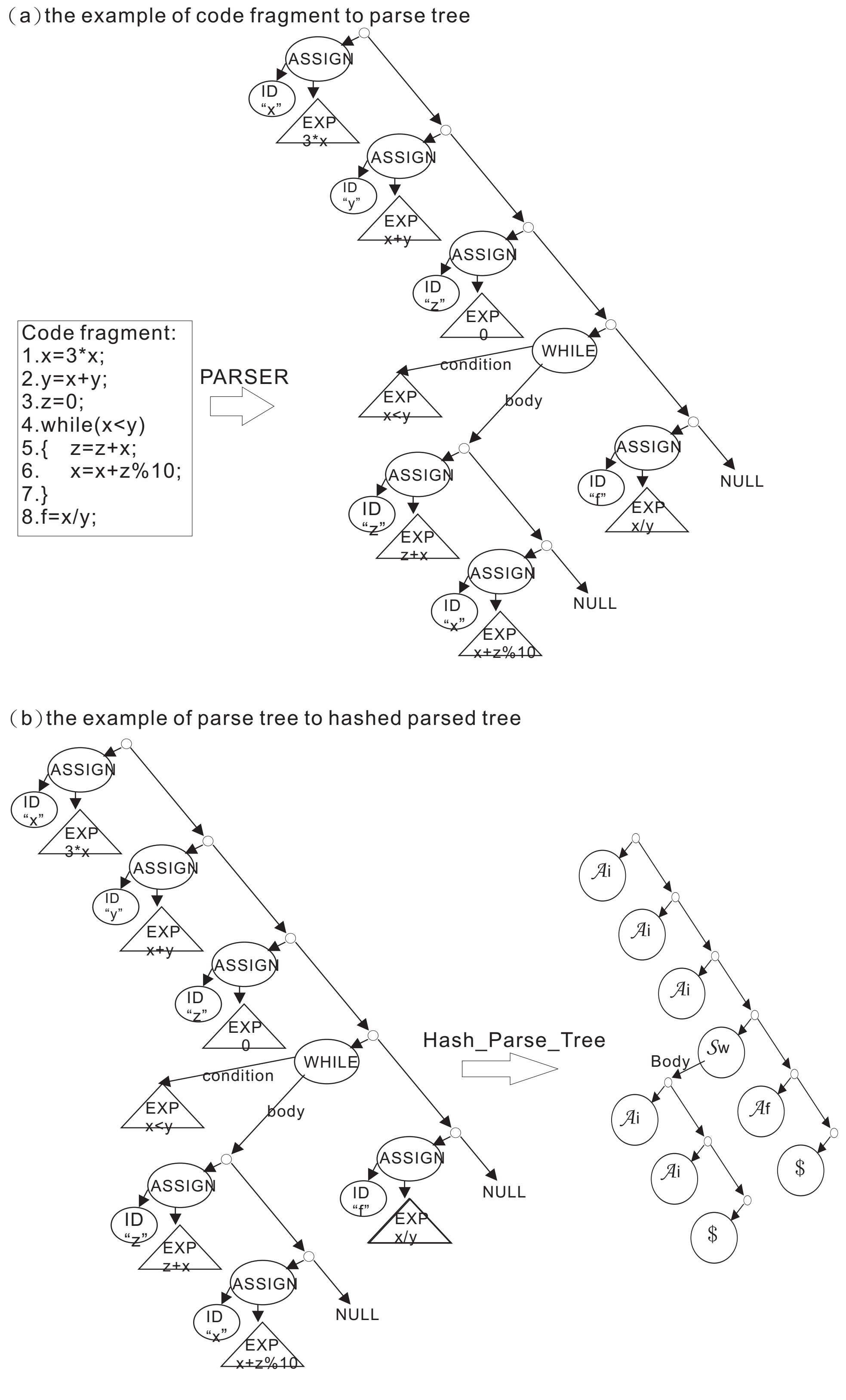


Fig. 8. The hashed parse tree for the example program.

Algorithm: Construct_Suffix_Tree

```
Input : $\mathcal{T}_h$
Output : Suffix Tree $\mathcal{T}_s$
Initialize $\mathcal{T}_s$ and Stack $\mathcal{S}$
Construct_Suffix_Tree($\mathcal{T}_h$)
{   sf_head = $\mathcal{T}_s$.root;
    stage = $\mathcal{T}_h$.root;
    head = $\mathcal{T}_h$.root;
    in_loop = 0;
    while(stage $\neq$ NULL)
    {   while(exists $n$ in sf_head.children such that
                        $\mathcal{C}_h(n$, head) is true)
        {   sf_head = $n$;
            record_location($n$, head);
            head = head.next;
        }
        Add_Suffix_tree(sf_head, head);
        sf_head = $\mathcal{T}_s$.root;
        stage = Next_Stage(stage);
        head = stage;
    }
}
```

Fig. 9. The Construct_Suffix_Tree algorithm

5.2 Constructing Suffix Tree

The second step in our framework of clone detection is to construct the suffix tree from the hashed parse tree. The idea of using suffix tree for clone detection is included in a few techniques of token-based clone detection [1,2,3,4,8,17]. We adapt the function for constructing a suffix tree in order to take a hashed parse tree as its input.

The algorithm that we devleop for constructing a suffix tree $\mathcal{T}_s$ from the hashed parse tree $\mathcal{T}_h$ is presented in Figure 9. The main idea is adding the substring of a string $\mathcal{S}$ of each stage into the suffix tree root, which the substring of each stage is a substring that was removed the first letter of the substring in the previous stage.

Function: Next_Stage

```
Next_Stage($p$)
{   switch($p$)
    {   case $\mathcal{A}_i$, $\mathcal{A}_c$, $\mathcal{A}_t$, $\mathcal{S}_f$ :   $p = p$.next;
                                    break;
        case $\mathcal{S}_i$ :   push($\mathcal{S}$, $p$.next);
                   push($\mathcal{S}$, $p$.else);
                   $p = p$.then;
                   break;
        case $\mathcal{S}_w$ :   push($\mathcal{S}$, $p$.next);
                   $p = p$.body;
                   break;
        case \$ :     $p = $ pop($\mathcal{S}$);
    }
    return $p$;
}
```

Fig. 10. The Next_Stage Function

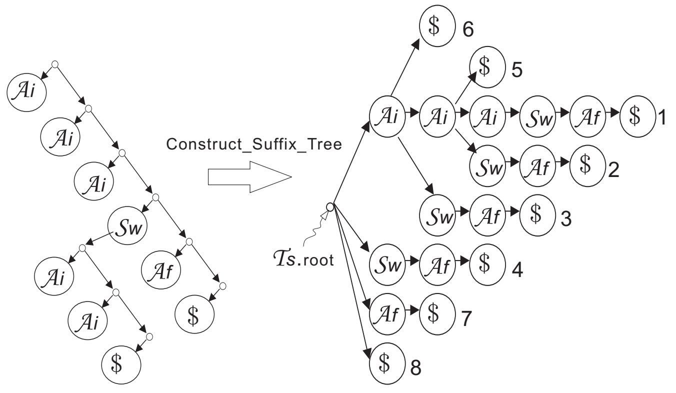


Fig. 11. The suffix tree example .

In the Construct_Suffix_Tree algorithm, we add the substrings for subtrees in the hashed parse tree into the suffix tree. In the case that the hash key k is $\mathcal{S}_i$ or $\mathcal{S}_w$, iteratively add each subtree of its statement body into the suffix tree.

In our algorithm for constructing the suffix tree, we use the compare functions $\mathcal{C}_h$ and $\mathcal{C}_e$ in Figure 7 to compare hash keys of the nodes h_1 and h_2 and check if they can be a parametrized match. If the keys of h_1 and h_2 both are $\mathcal{S}_i$ or $\mathcal{S}_w$, we recursively use the function $\mathcal{C}_h$ to compare the subtree of their body. The $\mathcal{C}_e$ function is used to compare the *condiftion* of two nodes if their hash

Algorithm: Clone_Detect

```
Input : $\mathcal{T}_s$
Output : a set of clones $\Theta$
Initialize $\Theta$, head = $\mathcal{T}_s$.root;
Clone_Detect(head)
{   for all $n$ in head.children
        if($n$.depth < $\boldsymbol{K}$)
            Clone_Detect($n$);
        else
        {   if($n$.count $\geqq$ $\boldsymbol{N}$)
            {   $\mathcal{C}$ = Creat_Clone();
                temp = $n$;
                while(temp.parent $\neq$ $\mathcal{T}_s$.root)
                    temp = temp.parent;
                Copy($\mathcal{C}$.location, temp.location);
                $\mathcal{C}$.depth = $n$.depth;
                Add $\mathcal{C}$ into $\Theta$
                Clone_Detect($n$);
            }
        }
}
```

Fig. 12. The Detect_Clone algorithm.

key both are $\mathcal{S}_w$ or $\mathcal{S}_i$ that is in *while* loop body. Hence the value of *in_loop* would add by 1 when comparing the loop body and reduce by 1 when going to the end of comparing. It is of note that the variable *in_loop* is a flag to check the current statement is in loop body or not, but this variable only affects the compare function $\mathcal{C}_h$ with $\mathcal{S}_w$, i.e it can't affect any stage in this step even though the stage is in the loop body. The algorithm of the auxiliary functions is shown in Figure 7, 10. Figure 11 shows the suffix tree that our algorithm constructs for the hashed parse tree in Figure 8.

5.3 Detecting Clones

The last step in our framework is computing the clone set of the program from the suffix tree. We take the suffix tree $\mathcal{T}_s$ as input to compute the set of clone class Θ. Since the small code fragments may not be of our interest, we allow users to set two threshholds for the clone sets to compute:

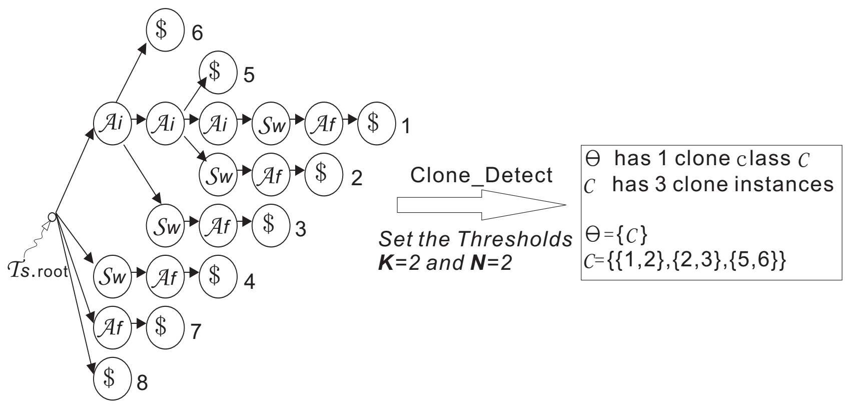


Fig. 13. The Detecting Clones example

- The number of statements in the code segment, $\boldsymbol{K}$, and
- The number of occurrances of the clone, $\boldsymbol{N}$.

The algorithm that we develop for detecting clones, called Clone_Detect, is shown in Figure 12. In the algorithm, we compute the distance of each node n from the root of $\mathcal{T}_s$ denoted as $n.depth$. It is clear that $n.depth$ is the number of statements on the path from the root to $\mathcal{T}_s$. And, we compute the number of leafs of the subtree rooted at n, denoted as $n.count$. It is clear that $n.count$ is the number of repetitions of the code segment. For each node n, if $n.depth > \boldsymbol{K}$ and $n.count > \boldsymbol{N}$, a parameterized clone instance of the path from the root of $\mathcal{T}_s$ to n is found. We use the algorithm to compute the suffix tree in Figure 11 and the clone set is $\{\{1,2\},\{2,3\},\{5,6\}\}$ as shown in Figure 13.

6 Discussion

Our parameterized clone deteion is based on the simple language $\mathcal{L}_s$ proposed in section 3. Then we would discuss how to extend our approach to fit in with practical programming language such as C language. And we also use a complete C program as an example to demonstrate how our approach is compared with Baker's and Baxter's. At last we have a comparison with a latter paper porposed by Evans et al.[7].

6.1 Extension

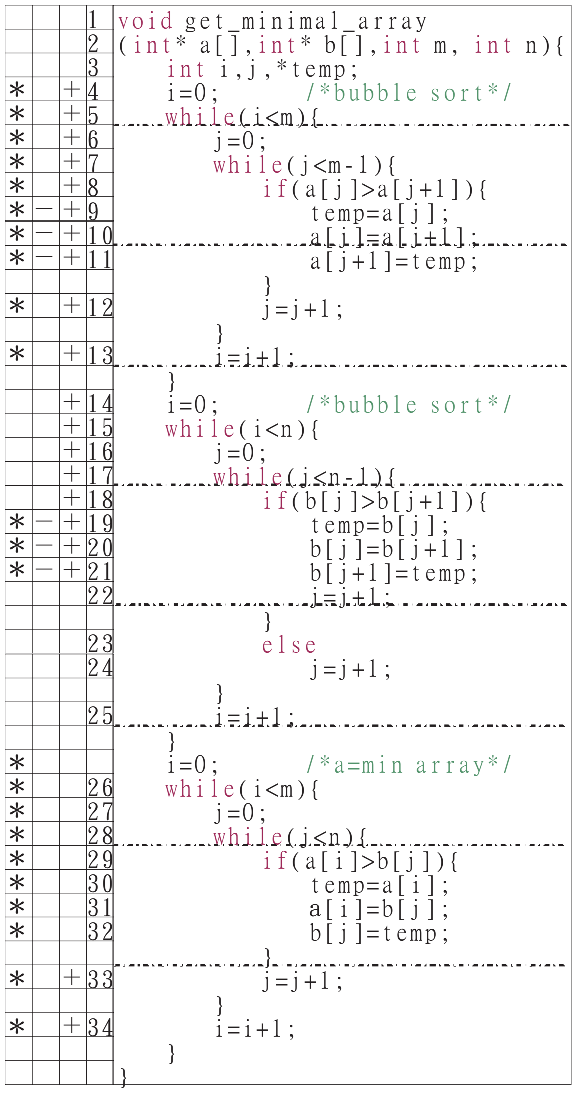
Our clone detection described in section 4 is only based on the simple language $\mathcal{L}_s$, generally too small, so we extend our approach to suit standard C language.

In the simple language $\mathcal{L}_s$, it only considers the necessary and basic functions such as if, *while*, and *assignment*. The $\mathcal{L}_s$ language must be included in standard C language, hence we need to handle more statements and types when extending our approach. The C language contains more data types than $\mathcal{L}_s$ language, but we can use the similar way to modify hashing, such as *double* type *assignment* hash into the key $\mathcal{A}_d$. And the C language also has those statements that are not contained in $\mathcal{L}_s$, in which statements could be classified into two types, one is nested structural statement such as for, *do*, or *switch*, another is single statement such as *declaration*, *break*, or *return*. For the first type, we give the new hash key for those statements and use the similar way to hash those statements and their nested body. For the second type, we choose the statements satisfied the definition of parameteried clone such as *break* or *continue* and give them the special hash keys $\mathcal{S}_b$ or $\mathcal{S}_c$; others in second type, those can't be transformed into procedures, would be hashed into the same default hash key "$\perp$" that would be considered as different key and always returned flase when computing the compare function.

6.2 An Example

Here we use a complete C program get_minimal_array to demonstrate how our approach is compared with Baker's approach and Baxter's approach. In Figure 14, the get_minimal_array program has 3 nested loops with a similar program structure, which is used to implement the bubble sort. In figure 14(a), we include a table marking the computation of the three approaches compared at the left of the program listing. The lines marked with a "*" sign are considered as in clones by our algorithm; and the lines marked with a "+" sign and a "-" sign are considered as in clones by Baxter's algorithm and by Baxter's algorithm, respectively.

In Figure 14(b), we calculate the number of lines that are considered in clones by each approach. In this example, there are 23 lines of code marked as in clones by our approach. And, there are 20 and 6 lines of code marked as in clones by Baker's and Baxter's approaches, respectively. It is clear that our approach can find more lines of code in the clones than Baker's and Baxter's approaches in this example. Baker's approach finds the number of lines of code in the clones to be close to ours, but the clones are almost syntatically incomplete. In the other words, the clones which are detected by Baker's approach (a)The Example



```
       1  void get_minimal_array
       2  (int* a[],int* b[],int m, int n){
       3      int i,j,*temp;
*   +  4      i=0;        /*bubble sort*/
*   +  5      while(i<m){
*   +  6          j=0;
*   +  7          while(j<m-1){
*   +  8              if(a[j]>a[j+1]){
* - +  9                  temp=a[j];
* - + 10                  a[j]=a[j+1];
* - + 11                  a[j+1]=temp;
                      }
*   + 12              j=j+1;
                  }
*   + 13          i=i+1;
              }
    + 14      i=0;        /*bubble sort*/
    + 15      while(i<n){
    + 16          j=0;
    + 17          while(j<n-1){
    + 18              if(b[j]>b[j+1]){
* - + 19                  temp=b[j];
* - + 20                  b[j]=b[j+1];
* - + 21                  b[j+1]=temp;
      22                  j=j+1;
                      }
      23              else
      24                  j=j+1;
                  }
      25          i=i+1;
              }
*             i=0;        /*a=min array*/
*     26      while(i<m){
*     27          j=0;
*     28          while(j<n){
*     29              if(a[i]>b[j]){
*     30                  temp=a[i];
*     31                  a[i]=b[j];
*     32                  b[j]=temp;
                      }
*   + 33              j=j+1;
                  }
*   + 34          i=i+1;
              }
          }
```

(b)Comparison

Criteria	Baker's Approach	Baxter's Approach	Our Approach
Marked with	+	−	*
Lines in clones	20	6	23
Lines in program	59%	18%	68%

Fig. 14. A complete example and the comparison.

almost can't be transformed into procedure calls for this example.

This example clearly shows that our goal in developing a new clone detection technique that finds more lines of code in clones. As long as the code fragments can be transformed into calls to a procedure with appropriate parameters, we consider the code segments as instances of a clone set.

6.3 Comparison with Evans' Approach

The *Asta* is an AST-based clone detection which is proposed recently by Evans et al.[7]. The purpose of Asta is also to find clones for procedural abstraction, but their viewpoint and method are both different from ours.

The viewpoint of Asta is to find larger clones for procedural abstraction, dissimilarly we find more clones for procedural abstraction that can reduce object code size. But this task is too heavy, hence we cut the task into two parts, one is to find all clones that can be transformed into proceural call, another is to analyze all found clones to produce optimizational procedural calls that can reduce code size efficiently. The secod part is our current research. It does not only need to consider the parameters of each procedural call affecting code size but also need to handle the overlapping of any clones. In Evans' paper, they did not describe the effect of parameters and the problem of overlapping clones.

The method of clone detection of Asta uses very large memory space to build the candidate patterns set $\prod$ and *clone table* in order to save the time complexity of comparing each sub-tree. Inversely the space of our approach just cost hashed tree and suffix tree, in which the size of each tree is near to parse tree.

7 Conclusion

In this paper, we discuss the concept of clone detection and procedure abstraction. We develop a new algorithm for clone detection from the viewpoint for procedure abstraction. Our algorithm consists of three steps: hashing the parse tree, constructing the suffix tree, and detecting the clones. For the clone set computed by our technique, each clone class may be transformed into a procedure, and each clone instance in the clone class may be replaced by a call to the procedure. We include the application of our approach and the comparison of our approach with the other approaches.

As the writing of this paper, the work on implementing an automatic tool for applying our new technique on C programs is at its final stage. The tool is designed to use with the gnu C compiler. Hence, our first direction in the near future is to finish the work on implementing the tool.

Our other future directions include developing a code size reduction technique by using automatic procedure abstraction based on the clones found by the technique presented in this paper. Since the patterns in the clones found may be very different, it requires further investigation on how to find the most efficient and effective way to abstract the clones into procedures.

References

[1] B.S. Baker ”A Program for Identifying Duplicated Code”, Journal of Computing Science and Statistics, vol. 24, pp. 49-57, March 1992.

[2] B.S. Baker ”A theory of parameterized pattern matching: algorithms and applications”, Proceedings of the twenty-fifth annual ACM symposium on Theory of computing STOC ’93, pp. 71-80, June 1993

[3] B.S. Baker ”On finding duplication and near-duplication in large software systems”, Proceedings of 2nd Working Conference on Reverse Engineering, pp. 86-95, July 1995

[4] B.S. Baker ”Parameterized pattern matching by Boyer-Moore-type algorithms”, Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms SODA ’95, pp. 541-550, January 1995

[5] I.D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier ”Clone detection using abstract syntax trees”, Proceedings of International Conference on Software Maintenance, pp. 368 - 377, November 1998 Analysis and Manipulation, pp. 128-135, 2004

[6] S. Ducasse, M. Rieger, and S. Demeyer ”A language independent approach for detecting duplicated code”, Proceedings of IEEE International Conference on Software Maintenance 1999. (ICSM ’99), pp. 109-118, August 1999

[7] W. Evans, C.W. Fraser, F. Ma, ”Clone Detection via Structural Abstraction”, Working Conference on Reverse Engineering, 14th WCRE’07, pp. 150-159, Oct. 2007.

[8] R. Giancarlo, “The suffix tree of a square matrix with applications,” Proceedings of 4^{th} Annual ACM-SIAM Conference on Discrete Algorithms (SODA), pp. 402-411, January 1993.

[9] S. Horwitz ”Identifying the semantic and textual differences between two versions of a program”, Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation PLDI ’90, pp. 234-245, June 1990

[10] J.H. Johnson ”Substring matching for clone detection and change tracking”, Proceedings of International Conference on Software Maintenance 1994, pp. 120-126, September 1994

[11] K. Kontogiannis, R. Demori, M. Bernstein, M. Galler, and E. Merlo ”Pattern Matching for Clone and Concept dectection”, Automated Software Engineering, vol. 3, no. 1, pp. 77-108, 1996

[12] R. Koschke, R. Falke, and P. Frenzel ”Clone Detection Using Abstract Syntax Suffix Trees”, Proceedings of the 13th Working Conference on Reverse Engineering (WCRE ’06), pp. 253- 262, October 2006

[13] T. Kamiya, S. Kusumoto, and K. Inoue ”CCFinder: a multilinguistic 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

[14] M. Kim, and D. Notkin ”Mining Software Repositories (MSR): Using a clone genealogy extractor for understanding and supporting evolution of code clones”, Proceedings of the 2005 international workshop on Mining software repositories MSR ’05, pp. 1-5, May 2005.

[15] J. Krinke ”Advanced slicing of sequential and concurrent programs”, Proceedings of the 20th IEEE International Conference on Software Maintenance, pp. 464-468, September 2004.

[16] M. Kim, V. Sazawal, D. Notkin, and G. Murphy ”An empirical study of code clone genealogies”, Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering ESEC/FSE-13, pp. 187-196, September 2005

[17] E. M. McCreight ”A Space-Economical Suffix Tree Construction Algorithm”, Journal of the ACM (JACM), vol. 23, no. 2, pp. 262-272, April 1976

[18] J. Mayrand, C. Leblanc, and E. M. Merlo ”Experiment on the automatic detection of function clones in a software system using metrics”, Proceedings of International Conference on Software Maintenance, pp. 244-253, November 1996

[19] Semantic Designs, Inc. “Clone Doctor: Software Clone Detection and Removal,” The web site at
http://www.semdesigns.com/.

[20] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue ”On detection of gapped code clones using gap locations”, Proceedings of the Ninth Asia-Pacific Software Engineering Conference, pp. 327-336, December 2002

[21] F. Van Rysselberghe, and S. Demeyer ”Evaluating clone detection techniques from a refactoring perspective”, Proceedings of 19th International Conference on Automated Software Engineering 2004, pp. 336-339, 2004

[22] T. A. Wagner, and S. L. Graham ”Incremental analysis of real programming languages”, Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation PLDI ’97, pp. 31-45, May 1997

[23] V. Wahler, D. Seipel, J. Wolff, and G. Fischer ”Clone detection in source code by frequent itemset techniques”, Proceedings of the Fourth IEEE International Workshop on Source Code

[24] C. Yung, and S-L. Chen. “High-Level Procedural Abstraction for Reducing Size of Object Code,” *Proceedings on the Internation Computer Symposium 2002*, Taiwan, December 2002.

[25] C. Yung and C-W. Wu ”A new approach to parameterized clone detection using abstract syntax tree”, CTHPC 2008.