A New Approach to Parameterized Clone Detection Using Abstract Syntax Tree

Chung Yung and Che-Wei Wu

Abstract—This paper describes a new approach of parameterized clone detection using abstract syntax tree. Since a considerable fraction of the source code of 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 clones detected into procedure calls is possible.

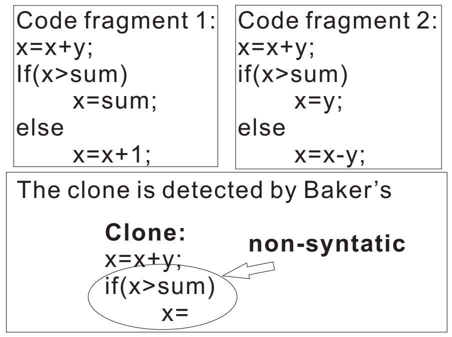
Index Terms—Clone detection, code clone, procedure abstraction, abstract syntax tree

1 Introduction

In this paper, we propose a new approach to *parameterized clone detection* to find out the clones which get to be replaced by procedure calls. In our front research, it can reduce the object code size through the procedural abstraction before code generation stage. The mainly idea of procedural abstraction is to abstract the identical parts in a program into a single procedure, hence we need a clone detection which is more powerful than other approaches that are not used for this purpose. This paper present simple and practical approach for detecting the parameterized clones which can be used by procedural abstraction.

Existing research[18] suggests that large software systems typically contain 10-25% redundant code (clones). This redundancy is caused by reusing code through copy-and-paste or programming the structural similarities code accidentally. If there are many clones in a program, it cause code sized large and unstructured in which we need large memory to store large object code and spend more cost to maintain the unstructured code. Hence, detecting and refactoring the clones are very important topics.

In the meantime, detection and refactoring of clones by procedure calls promises decreased program maintain cost corresponding to the reduction in code size. There are many types of clone detections in the research area of software engineering, but these approaches do not aim at procedural abstraction. It implies that the constrain of clones is too serious or the semantics of some detected clones is not completed. For example, there are two code fragments in Figure1. It shows the detected clone is incomplete and syntactically incorrect. Since there are above problems in existent clone detections, we propose a abstract-syntax-tree-based approach to improve these problems.



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

Fig. 1. The Syntactically Incorrect Example.

According to clone detection techniques[20], it can be divided into *string-based*[6], [8], *token-based*[3], [11], or *abstract-syntax-tree-based(AST)*[5], [9], [13], [16]. String-based clone detection divides the program in a number of strings such as lines and whole lines compared to each other textually[6]. Token-based clone detection transforms the program into a token stream and then search the similar token sequence by suffix tree. AST-based clone detection finds similar subtree after building a parse tree. By our observation, the existent clone detections find the similar code fragments as clones, but our intention is different from these. We are interested in those fragments that perhaps can

• *C. Yung and C.W. Wu are with the Department of Computer Science and Information Enginnering, National Dong Hwa University, Taiwan, R.O.C.*
E-mail: yung@mail.ndhu.edu.tw
E-mail: m9521038@em95.ndhu.edu.tw

Manuscript submitted march 7, 2007.

```
(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.

be transformed into procedure calls. For example in Figure2, there are 6 code fragments which are little different from each other. According to existent clone detections, they could find out the clone which is {(1),(2)} or {(1),(2),(3)}. But we think about that these 6 code fragments all can be transformed into a single procedure call. And we describe how to transform in section 3.

The existent clone detections can not detect the clones that are interesting for us, hence we propose a new approach to detect those clones that can be transformed into procedure calls. Our approach is based on AST, but we do not compare each subtree. Since the tree matching is inefficient, we use the concept of suffix tree to detect interesting clones. And we describe the detail of algorithm in section 4.

2 Related Works

For clone detection techniques, there are many different approaches to find out clones, which the definition of clone are also varied. Our approach is like AST-based technique, but we use the concept of suffix tree structure which is used in token-based technique. Hence we introduce the AST-based approach that is proposed by Baxter and the token-based approach that is proposed by Baker in this section.

2.1 Baxter's AST-based Approach

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

(a)Basic Algorithm

```
1.Clones=Ø
2.For each subtree i
      If mass(i)>=Threshold
      Then hash i to bucket
3.For each subtree i and j in the same bucket
      If CompareTree(i,h)>SimilarityThreshold
      Then { For each subtree s of i
                  If IsMember(clones,s)
                  Then RemoveClonePair(clones,s)
             For each subtree s of j
                  If IsMember(clones,s)
                  Then RemoveClonePair(clones,s)
             AddClonePair(clones,i,j)
```

(b)Sequence Detection Algorithm

```
Sequence Detection Algorithm
1. Build the list structures describing sequences
2. For k = MinimumSequenceLengthThreshold
           to MaximumSequenceLength
3. Place all subsequences of length k
   into buckets according to subsequence hash
4. For each subsequence i and j in same bucket
   If CompareSequences (i,j,k)> SimilityThreshold
   Then {  RemoveSequenceSubclonesOf (clones,i,j,k)
           AddSequenceClonePair (clones,i,j,k)
        }
```

Fig. 3. The Baxter's Algorithm.

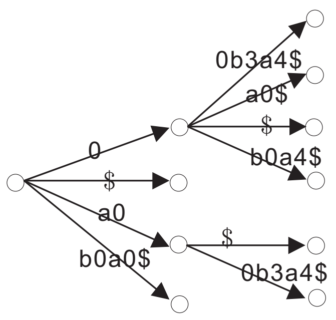
But computing the similarities of all subtree pairs is not efficient, which complexity of computation is O(N^3) for the number N of nodes of AST. In order to solve this serious problem, Baxter have proposed that using hashing function to hash subtrees into some buckets if the mass of subtree exceeds the mass threshold. Give the basic algorithm as Figure 3.a.

The single subtree clones were found by above algorithm, but the subtree sequence clones can not be detected. Baxter build a list structure where each list is associated with a sequence in the program, and stores the hash codes of each subtree element of the associated sequence. The Figure 3.b gives the sequence clone detection algorithm which compares each pair of subtree sequence clone if the length of clone sequence exceeds the given threshold. This idea is similar to subtree clone detection, which compares all pair of subtree or all pair of subtree sequence, so this algorithm also has a hash function to hash the sequence into some buckets.

Baxter's clone detection used the abstract syntax tree and compared each subtree or subtree sequence to find out exact and near-miss clones. Because this approach is a tree match, it has higher complexity than token-based approach. Then we describe the token-based approach that is proposed by Baker.

2.2 Baker's Token-based Approach

The approach is based on suffix trees which is used for efficient string pattern match. Because it is a string



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

Fig. 4. Example of p-suffix tree of p-string $\mathcal{S}$.

pattern match, it has good performance but it implies the found clones illogical.

The idea of token-based clone detection is proposed by Baker[1], [2], [3], [4]. Baker's approach detected the exact and parameterized match 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 parameters 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 symbol". A code fragment such as x=x+y is first transformed into P=P+P and a list x,x,y; then a non-parameter symbol is generated to represent the 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 that is called a *parameterized string* or *p-string*). In order to detect p-match from p-string, this approach also needs to encode the parameter symbols such way as followed. 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 $\$$ represent non-parameter symbols, and x and y represent parameter symbols, a p-string *axybxay$* would be encoded as *a00b3a4*.

After the lexical analysis, ***dup*** which is a detecting parameterized match algorithm 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 $\mathcal{S}$, for above example $\mathcal{S}$, *axybxay$*, $\mathcal{S}$ has 8 suffix substrings that are *axybxay$*, *xybxay$*, *ybxay$*, *bxay$*, *xay$*, *ay$*, *y$*, *$*. A suffix tree is a representation of a string as a trie where every suffix is present through a path from the root to a leaf. For any two leaf, we have two distinct paths from root to the leaf. And if there is a common prefix shared path, the shared path is a p-match substring of p-string, which substring is a clone. Figure 4 shows the example constructing suffix tree 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 there are linear algorithms to construct suffix tree which are proposed by McCreight[15]

3 A Sample Language

This section describes a simply sample program language for our algorithm. The sample language simplifies the standard $\mathcal{C}$language, and we call it $\mathcal{C}_s$ language. The $\mathcal{C}_s$ language is used to describe our algorithm simply and clearly, because we do not need to care those nonessentials part of $\mathcal{C}$ language. The following describes the token and syntax of $\mathcal{C}_s$ language.

3.1 Token List of $\mathcal{C}_s$

A lexical scanner for $\mathcal{C}_s$ should be recognized as tokens from $\mathcal{C}_s$ language that simplifies $\mathcal{C}$ language. Listed below are main definitions of the pattern of lexical tokens that are legal for $\mathcal{C}_s$ language.

1) int {int}
2) char {char}
3) float {float}
4) = {assign}
5) == {eq}
6) > {gt}
7) < {lt}
8) >= {ge}
9) <= {le}
10) if {if}
11) else {else}
12) while {while}
13) return {return}
14) [a-zA-z]+[0-9a-zA-Z]* {id}
15) [0-9]+(.[0-9]+)? {number}

3.2 Syntax of $\mathcal{C}_s$

Listed following are the context-free grammar for $\mathcal{C}_s$ language.

1) $\mathcal{C}_s$: GDECLIST
2) GDECLIST: GDEC GDECLIST | GDEC
3) GDEC: ODECL | FUN
4) ODEC: TYPE DECLIST ;
5) TYPE: int | char | float
6) DECLIST: DEC , DECLIST | DEC

7) DEC: NAME
8) NAME: id
9) FUN: TYPE NAME (ARGLIST) COMSTMT
10) ARGLIST: ARG , ARGLIST | ARG
11) ARG: TYPE NAME | (null)
12) COMSTMT: { STMTLIST }
13) STMTLIST: STMT STMTLIST | STMT
14) STMT: COMSTMT | ODEC | EXPLIST | IFSTMT | WHILESTMT | RESTMT
15) EXPLIST: EXP , EXPLIST | EXP | ()null)
16) EXP: PRIM | PRIM assign PEXP
17) PEXP: PEXP OP1 TERM | TERM
18) TERM: TERM OP2 FACTOR | FACTOR
19) FACTOR: PRIM
20) PRIM: NAME | NAME(EXPLIST) | number
21) OP1: + | -
22) OP2: * | / | %
23) IFSTMT: if (REEXP) STMT ELSESTMT
24) ELSESTMT: else STMT | (null)
25) REEXP: EXP RELOP EXP | EXP
26) RELOP: eq | gt | ge | lt | le
27) WHILESTMT: while (REEXP) STMT
28) RESTMT: return | return EXP

4 Parameterized Clone

In this paper, we are interested in the code fragments that perhaps can be transformed into procedure calls, and called parameterized clones. Hence, we give the definition of parameterized clone here, and we use the definition as the vest 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 $\mathcal{F}_1, ..., \mathcal{F}_k \in \mathcal{F}$ with appropriate parameters. In such a case, we call p_i is a clone instance of clone class $\mathcal{C}$, where $1 \leq i \leq k$.**

We say that Θ is a set of *parameterized clones class* and a *parameterized clone class* is a set of *clone instances* that can transformed into a single procedure call. And then we discuss about what kind of code fragments can be transformed into procedure calls.

Code clones are generally know as duplicated code fragments through copy-and-paste in a software system. In convention, existent clone detections are designed to compare two code fragments similar or not by computing similarities or matching patterns, such as Baxter's approach or Baker's. But we consider two code fragments as parameterized clone if they can be transformed into a single procedure call. For an example in the Figure 2 of section 1, there are 6 code fragments. The clone class {(1),(2)} is detected by Baker's approach in which code fragments are p-match, and the clone class {(1),(2),(3),} is detected by Baxter's

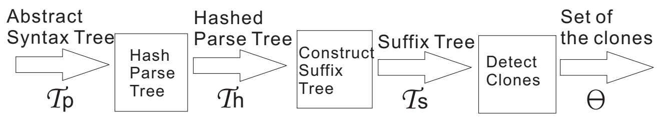


Fig. 5. The Framework of Our Algorithm.

approach in which the nodes of each code fragments are the same. Both of them do not deem the set {(1),(2),(3),(4),(5),(6)} that is a clone class. We consider the set {(1),(2),(3),(4),(5),(6)} as a parameterized clone class because each clone instance in the set can be transformed into a single procedural call $\mathcal{F}$, such as $\mathcal{F}$(&X, A+B, &Z, A-B), $\mathcal{F}$(&X, P+Q, &Z, P-Q), $\mathcal{F}$(&X, P+Q, &Z, A-B), $\mathcal{F}$(&X, P+1, &Z, P-1), $\mathcal{F}$(&X, P*Q, &Z, P/Q), and $\mathcal{F}$(&X, 5, &Z, 10). Although the facades of (3), (4), and (5) are different, but each code fragments is a statement sequence that is *integer-assignment*($\mathcal{A}_i$) by *integer-assignment*($\mathcal{A}_i$). Hence we call the set of these 6 fragments parameterized clone class, it is similar to *char-assignment*($\mathcal{A}_c$) and *floating-assignment*($\mathcal{A}_f$). Generally speaking, if there are two code fragments which have the same kind of statement sequence, we may regard these two fragments as parameterized clone class.

Moreover, we consider loop statements, such as *while*, and selection statements, such as *if*, in a program. For the two kinds of statements, we ignore their *condition* and *body*, and call them $\mathcal{S}_w$*(while)* and $\mathcal{S}_i$*(if)*. For example, two selection statements, $if(x > y)then...else...$ and $if(a <= b)then...else....$ These two statements both call $\mathcal{S}_i$ because we think about that the values of the condition of $\mathcal{S}_i$ are boolean values which we consider as parameters of the made procedure, such as $\mathcal{F}$(x>y, ...) and $\mathcal{F}$(a<=b, ...). The bodies of loop statements or selection statements are ignored at first, but we would compare each body at constructing suffix tree.

At last we consider *non-return-value* function calls. If two function calls are regarded as sameness, their function name must be the same. Hence we give all function calls the same key $\mathcal{S}_f$, but we save the information of the function name into this key. Altogether, we simply classify all statements into 6 types that are $\mathcal{A}_i$, $\mathcal{A}_c$, $\mathcal{A}_f$, $\mathcal{S}_w$, $\mathcal{S}_i$, and $\mathcal{S}_f$. There are not only 6 types of statements in a practical language, such as *for loop*, but we can use the similar way to classify others types. Hence we support a simple language that only allows these 6 types in this paper. But we would implement our approach in a practical language C.

5 Clone Detection

In this section, we describe our algorithm in detail. We proposed a new approach for clone detection, which is

```
Algorithm: Hash_Parse_Tree
Input: Parse Tree $\mathcal{T}p$
Output: Hashed Parse Tree $\mathcal{T}h$
Initialize $\mathcal{T}h$, p=$\mathcal{T}p$.root, and h=$\mathcal{T}h$.root
Hash_Parse_Tree(p,h)
{       h.key = $\mathcal{H}(p)$;
        switch(h.key)
        {  CASE $\mathcal{S}f$ :     h.name = p.func_name;
                                     break;
           CASE $\mathcal{S}w$:      h.body = create_node();
                                     Hash_Parse_Tree(p.body, h.body);
                                     break;
           CASE $\mathcal{S}i$:      h.then = create_node();
                                     Hash_Parse_Tree(p.then, h.then);
                                     If(p.else)
                                     {      h.else = create_node();
                                            Hash_Parse_Tree(p.else, h.else);
                                     }
                                     break;
           //$\mathcal{A}i$, $\mathcal{A}c$, and $\mathcal{A}f$ do nothing
        }
        h.next = creat_node();
        if(p.next)
              Hash_Parse_Tree(p.next, h.next);
        else
              h.next.key = $;
}
```

Fig. 6. The Hashing Parse Tree Algorithm.

a AST-based clone detection and use the suffix tree to find out clones. In the other word, we use the abstract syntax tree to construct the suffix tree to find out clones.

Our algorithm can be divided into three steps. First step, we hash the parse tree of the program. We use the hash function to classify all statements into six keys, the input of this step is the parse tree of a program $\mathcal{T}_p$ and the output is the hashed parse tree $\mathcal{T}_h$. Step 2, we construct the suffix tree. In this step, there are two comparison function, which one compares the single node of statement, another compares the subtree of statement body. The input of this step is $\mathcal{T}_h$ and the output is the suffix tree $\mathcal{T}_s$. Step 3, we find out all clones that are satisfied the thresholds that can be defined by user. In this step, we set the threshold of number of statements and the threshold of repetition. The input is the suffix tree $\mathcal{T}_s$ and the output is a set of clone classes Θ. The framework of our algorithm shows in Figure 5.

We describe the first step in section 4.1, the second step in section 4.2, and the last step in section 4.3.

5.1 Hashing Parse Tree

In this section, we describe the hash function and the hash parse tree algorithm. The hash function $\mathcal{H}$ hash the statement nodes of parse tree to the hash keys(values) $\mathcal{K}$. We denote the hash functions $\mathcal{H}$:$\mathcal{N} \rightarrow \mathcal{K}$, where $\mathcal{N}$ is the set of statement nodes of parse tree, and $\mathcal{K}$ is the set of hash keys that are $\mathcal{A}_i$, $\mathcal{A}_c$, $\mathcal{A}_f$, $\mathcal{S}_i$, $\mathcal{S}_w$, and $\mathcal{S}_f$. The hash function is shown as follow.

Let $\mathcal{H}$ be a hash function, and $n \in \mathcal{N}$, such that

$$\mathcal{H}(n) = \begin{cases} \mathcal{A}_i & \text{if } n.\text{kind=ASSIGN and } n.\text{type=INT} \\ \mathcal{A}_c & \text{if } n.\text{kind=ASSIGN and } n.\text{type=CHAR} \\ \mathcal{A}_t & \text{if } n.\text{kind=ASSIGN and } n.\text{type=FLOAT} \\ \mathcal{S}_i & \text{if } n.\text{kind=IF} \\ \mathcal{S}_w & \text{if } n.\text{kind=WHILE} \\ \mathcal{S}_f & \text{if } n.\text{kind=FUNC\_CALL} \end{cases}$$

We also propose the Hash_Parse_Tree algorithm as Figure 6. The algorithm is using $\mathcal{H}$ to hash each statement of the program parse tree. It stores the information of the function name into the node if the key of the node is $\mathcal{S}_f$. And if the key is $\mathcal{S}_w$ or $\mathcal{S}_i$, it hashes the body of $\mathcal{S}_w$ or $\mathcal{S}_i$ by recursion. Last we set the key $ for marking the end of statement list. Show the example of hashing parse tree in Figure 7.

5.2 Constructing Suffix Tree

In this section, we describe how use the hashed parse tree $\mathcal{T}_h$ to construct the suffix tree $\mathcal{T}_s$ and show the algorithm. We use the concept of suffix tree while its input is a string which is different from the input of our algorithm. The main idea of constructing suffix tree is adding the substring of 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 latter of the substring of the front stage. Hence we abstract the concept for tree input. In our algorithm, we add the subtree of hashed parse tree of each stage into suffix tree root, which the root of subtree of each stage is the next pointer of the root of subtree of front stage except $\mathcal{S}_i$ and $\mathcal{S}_w$ nodes. When we consider the next stage which hash key k of root is $\mathcal{S}_i$ or $\mathcal{S}_w$, we would push the next pointer to stack and make the root of next stage be the first node of its body. And we would pop the pointer to be next stage root from stack when we get the symbol $ which implies that we get end of body of $\mathcal{S}_i$ or $\mathcal{S}_w$.

When we add the subtree of each stage into the suffix tree, we use the function $Compare\_Node(M, N)$ to decide the nodes, M and N, sharing the same node or not by comparing their keys. If the keys of M and N both are $\mathcal{S}_i$ or $\mathcal{S}_w$, we use the function $Compare\_Sub(m, n)$ to compare the subtree of their body. Then we show the algorithm and the three major auxiliary function of the algorithm as Figure 8, 9. And the example of constructing suffix tree show in Figure 10.

5.3 Detecting Clones

In this section, we describe how to find out the parameterized clones from the suffix tree $\mathcal{T}_s$ and show the algorithm. The input is the suffix tree $\mathcal{T}_s$, and the output is a set of clone class Θ. We set the threshold of number of statement $\boldsymbol{K}$ and the threshold of number of repetition

(a)

```
Algorithm: Construct_Suffix_Tree
Input: $\mathcal{T}_h$
Output: Suffix Tree $\mathcal{T}_s$
Initialize $\mathcal{T}_s$ and stack $\mathcal{S}$
sf_head = $\mathcal{T}_s$.root;
stage = $\mathcal{T}_h$.root;
head = $\mathcal{T}_h$.root;
while(stage != NULL)
{       while( exists n in sf_head.children such that
                        Compare_Node(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;
}
```

(b)

```
Function: Compare_Node
Compare_Node(N1, N2)
{  if(N1.key != N2.key)
        Return false;
   else
   {    switch(N1.key)
        CASE $\mathcal{A}_i$,$\mathcal{A}_c$,$\mathcal{A}_f$ :        return true;
                                    break;
        CASE $\mathcal{S}_f$ :    if(N1.name == N2.name)
                          return true;
                      else
                          return false;
                      break;
        CASE $\mathcal{S}_i$ :    if( Compare_Sub(N1.then, N2.then) )
                          return Compare_Sub(N1.else, N2.else);
                      else
                          return false;
                      break;
        CASE $\mathcal{S}_w$:   if( Compare_Sub(N1.body, N2.body) )
                          return true;
                      else
                          return false;
                      break;
        default :       return false; // $
   }
}
```

Fig. 8. The Constructing Suffix Tree Algorithm and the Compare Node Functions.

(a)

```
Function: Compare_Sub
Compare_Sub(S1, S2)
{   if(S1.key == $ && S2.key == $)
       return true;
    if(Compare_Node(S1, S2))
       return Compare_Sub(S1.next, S2.next);
   else
       Return false;
}
```

(b)

```
Function: Next_Stage
Next_Stage(p)
{       switch(p)
        {CASE $\mathcal{A}_i$,$\mathcal{A}_c$,$\mathcal{A}_f$,$\mathcal{S}_f$ : 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 $ :        pop($\mathcal{S}$);
                          break;}
        return p;
}
```

Fig. 9. The Compare Subtree and the Next Stage Functions .

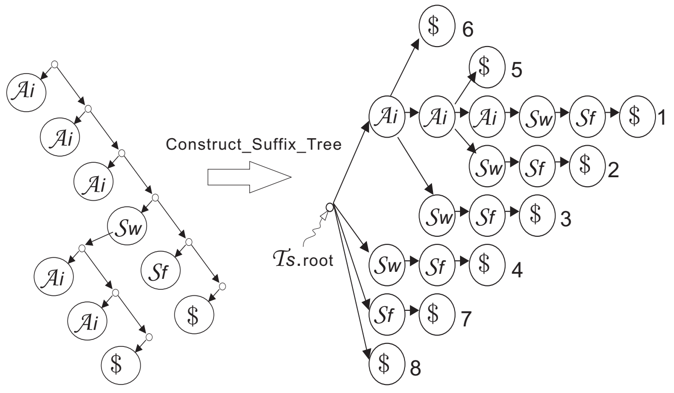


Fig. 10. The Suffix Tree Example .

N. And we consider the distance of each node n from the root of $\mathcal{T}_s$ to n, which is the number of statement, and consider the number of leafs of the subtree rooted at n, which is the number of repetition. For each node n, if the number of statement of n and the number of repetition of n both exceed the threshold ***K*** and ***N***, the path from the root of $\mathcal{T}_s$ to n is a parameterized clone. We show the algorithm in Figure 11(a) and the example in Figure 11(b).

6 An Example

We now consider a complete program code which is a function to get a minimal array from two arrays. In this function, there are 3 nested loops that have similar structure, in which two are used for bubble sorting. In figure 12(a), the right side is the program and the left side is a marking table that marks the lines as clone by manually using three different approaches which are Baker's(+), Baxter's(-), and ours(*). we calculate the clone line count and rate in Figure 12(b) according to the marking table.

In Figure 12(b), we can see our approach detect the largest clone in this program code, since we can find out the sequence line 32 to line 43 as parameterized clone that can't be detected by other's approaches. And we can see the sequence line 17 to line 24 that is only detected by Baker's approach, but this sequence doesn't (a)Algorithm

```
Algorithm: Clone_Detect
Input: $\mathcal{T}_s$
Output: a set of clones $\Theta$
Initialize $\Theta$
Head = $\mathcal{T}_s$.root;
Clone_Detect(head)
{      FORALL n in head.children
            If(n.depth < K)
                  Clone_Detect(n);
            else
            {     if(n.count >= N)
                  {     $\mathcal{C}$ = Creat_Clone();
                        tmp=n;
                        while(tmp.parent != $\mathcal{T}_s$.root)
                              tmp = tmp.parent;
                        Copy($\mathcal{C}$.location, tmp.location);
                        $\mathcal{C}$.depth = n.depth;
                        Add $\mathcal{C}$ into $\Theta$
                        Clone_Detect(n);
                  }
            }
      return;
}
```

(b)Example

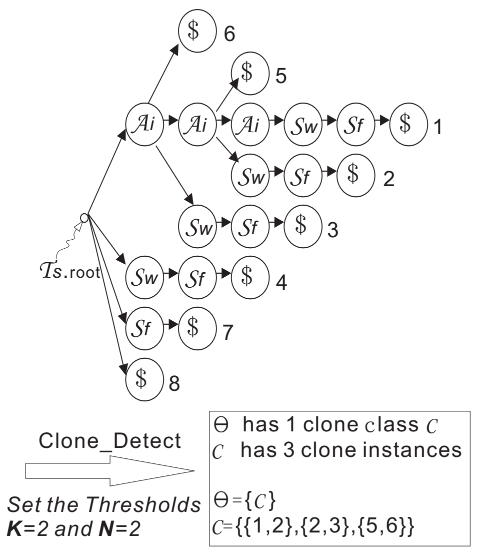


Fig. 11. The Detecting Clone Algorithm and Its Example.

have complete syntax since it doesn't include right braces for *while* and *if* statements. In other words, the clone detected by Baker's approach is a bad clone. It is similar to the sequence line 4 to line 11 that is detected by Baker's.

For our approach, we find out 23 lines program code as clone, in which it has 2 clone classes and 5 clone instances. One clone class includes 3 clone instances which are line 9 to 11, line 22 to 24, and line 37 to 39; another includes 2 clone instances which are line 4 to 15 and line 32 to 43. By the observation, we can see that these two clone classes have the intersection which

(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             }
*      13             j=j+1;
       14         }
*      15         i=i+1;
       16     }
     + 17     i=0;        /*bubble sort*/
     + 18     while(i<n)
     + 19     {   j=0;
     + 20         while(j<n-1)
     + 21         {   if(b[j]>b[j+1])
*  - + 22             {   temp=b[j];
*  - + 23                 b[j]=b[j+1];
*  - + 24                 b[j+1]=temp;
       25                 j=j+1;
       26             }
       27             else
       28                 j=j+1;
       29         }
       30         i=i+1;
       31     }
*      32     i=0;        /*a=min array*/
*      33     while(i<m)
*      34     {   j=0;
*      35         while(j<n)
*      36         {   if(a[i]>b[j])
*  - + 37             {   temp=a[i];
*  - + 38                 a[i]=b[j];
*  - + 39                 b[j]=temp;
       40             }
*      41             j=j+1;
       42         }
*      43         i=i+1;
       44     }
       45  }
```

(b)The manual result

Criteria	Baker's Approach	Baxter's Approach	Our Approach
Marked with	+	−	*
Lines in Clones	19	9	23
Lines in Program	42%	20%	51%

Fig. 12. A Complete Example and the Result.

causes that we only can choose one clone class to make the procedure. How to get the best clone class to make procedure is a important issue, hence we leave this topic for future work.

This example clearly shows that our intention of clone detection is different from other's. If only code fragment can be transformed into procedure call, we consider it as clone instance.

7 Conclusion

First, we proposed the different concept of the clones. For any two code fragments, if they are parameterized clone, these two fragments must can be replaced by one procedure. It is very different from traditional concept of clone, which clone is exact match or parameterized match.

Second, we proposed a hashing function such that this function can hash a statement to a key, which implied the parse tree to be simple. The hashing function can hash the same type statements into the same key no matter how many operands they have. Hence, we can detect the clones which are larger than others' detected.

Final, we used the suffix tree to find the common sequence for tree structure. It implies that we do not need to use the similarity function to compare two fragments that are similar or not. Comparing the similarity is a heavy task because it needs to compute all nodes of any two trees no matter the subtrees was already computed or not.

For reducing code size, the parameterized clones can be transformed into procedural calls, but how to find the most efficient way that transform these clones is a continuous research. This problem is not easy, we need to consider the cost of making procedural calls and the cost of each parameters. Also, if a clone instance is added an independent statement, it produce the problem "this instance is in original clone or not". These issue will discuss in our future research.

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] 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

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

[9] 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

[10] 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

[11] 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

[12] 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.

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

[14] 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

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

[16] 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

[17] R. W. Scheifler, and J. Gettys "The X window system", ACM Transactions on Graphics (TOG), vol. 5, no. 2, pp. 79-109, April 1986

[18] Semantic Designs, Inc. The web site at http://www.semdesigns.com/

[19] 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

[20] 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

[21] 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

[22] 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

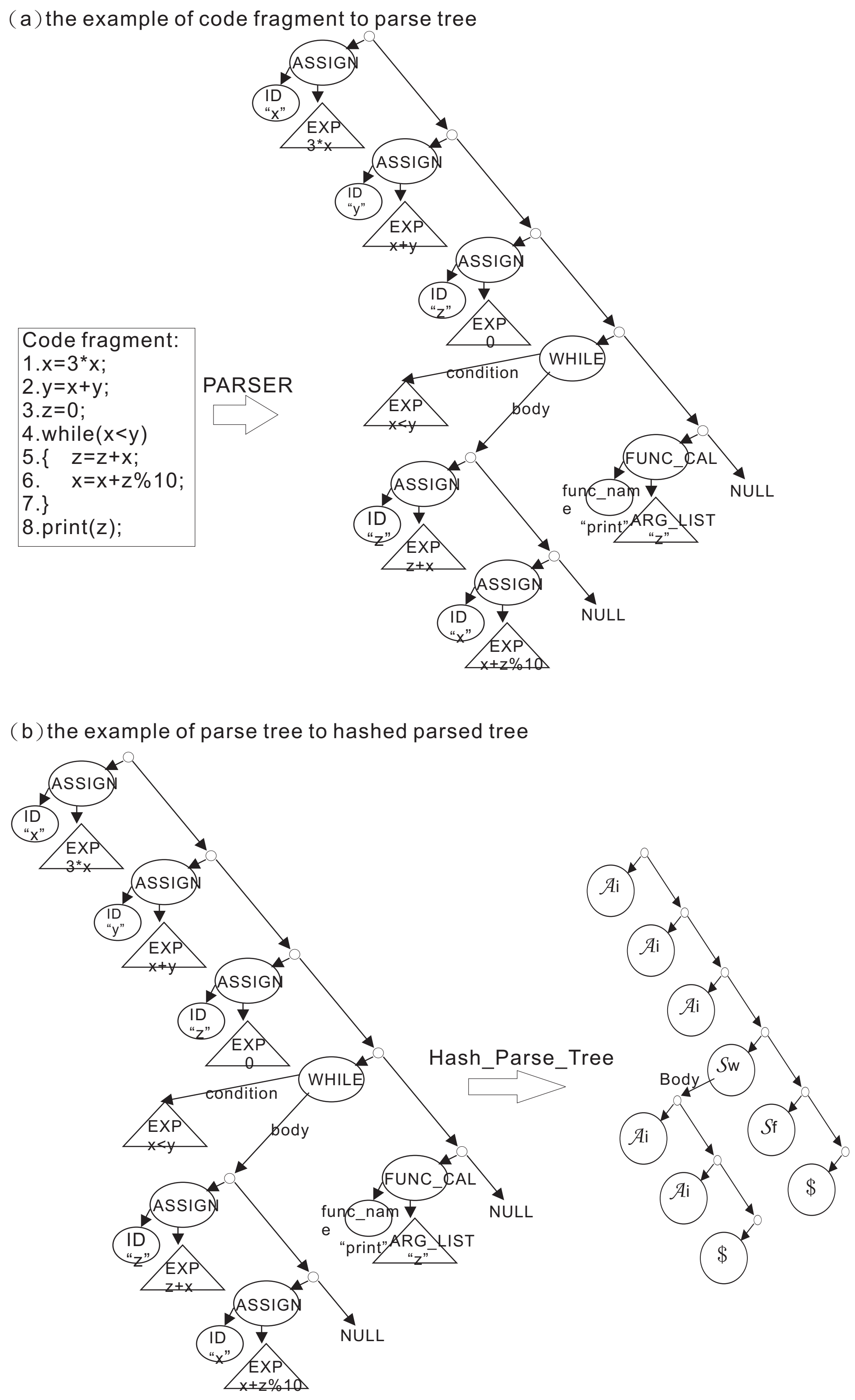


Fig. 7. The Hashed Parse Tree Example.