Mining of Source Code Concepts and Idioms

An Approach based on Clone Detection Techniques

Torsten Görg
Dept. of Programming Languages and Compilers
Institute of Software Technology
University of Stuttgart, Germany
torsten.goerg@informatik.uni-stuttgart.de

***Abstract*—This paper introduces a new view on program source code with a focus on code clone information. An algorithm is presented that transforms source code into an equivalent representation which expresses code redundancies as hierarchical clone classes explicitly. This representation supports program comprehension by pointing out arbitrary programming idioms and the frequencies of their occurrences in the program code. In contrast to most other code clone detection techniques looking for the largest clone fragments only the algorithm shown here provides clone classes for all fragment granularities and include relationships between the clone classes. This makes visible the full range from small idioms that occur frequently in the code to large fragments that are rarely used. Furthermore the provided code representation is specially designed to express parameterized clones as most programming idioms are not an adhesive code block but rather a subtree taken form the middle of an AST, e.g., the concept of a foreach loop without the loop body. The clone information condensed in the representation form shown here can either be used to support idiom-based program understanding. Or it can be used to refactor the most frequent idioms as new language concepts in a DSL in order to compact a given amount of source code.**

***Index Terms*—Code clones, programming idioms, clone classes, clone detection, clone representation**

I. Introduction

Most code clone detection techniques developed so far have the intention to support software maintenance. For many code clones it is important to do modifications and corrections on all instances of that clone in a given system consistently [1]. Clone detection tools have been created to ensure that none of these clone instances are missed. The tools detect all clones automatically and provide their search result as a set of clone pairs or clone classes [1]. Sometimes clone information is used to eliminate clone redundancy by introducing additional abstractions.

Our assumption is that clone detection techniques can also be used to support program comprehension. Many case studies have shown that large software system can contain huge amounts of cloned code. We can expect that in a usual system between 13% and 20% of the code is cloned [2]. Some COBOL systems even have a cloned code rate of about 50% [4]. This results in higher costs for manual program comprehension because the more code a program encompasses the more effort is usually required to understand it. If one knows which code fragments are redundantly repeated and where the repetitions are you just have to comprehend these fragments once and then you can apply this understanding to all clones of that code fragments.

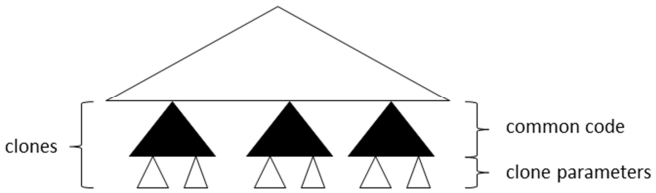

Fig. 1. Schematic parameterized clones in an AST (taken from [5])

When looking at code clones it is important to be aware that code clone is not a uniquely defined term [1]. Usually several clone types are differentiated. The main question is under which condition two code fragments are viewed as identical or similar. Most available clone detection approaches concentrate on clones that are adhesive code blocks. This means that such a block does not contain gaps that exclude code parts from a clone fragment. The percentages mentioned above are also based on clones of that kind. A more general category of clones is called type-3 clones [1]. This clone type allows some differences between code fragments that build a clone pair. But usually the differences are restricted to minor modifications like adding, removing, or replacing a code line.

A further generalization of type-3 clones is to consider parameterized clones[1] as introduced in [5]. Clone fragments differ in one or more parameters, and each parameter can be of any size. Projected on an AST an adhesive clone fragment is a complete subtree whereas a parameterized clone fragment is located in the middle of an AST and is characterized by an upper and a lower horizontal cut in the AST (see Fig. 1). A C++ code example is given in Fig. 2. The example contains two loops both of which iterate all elements of a container. The loops are instances of an idiom that implements the well-known foreach concept. Here the iterated container together

[1] This is different from the definition in [1]. Roy and Cordy specify parameterized clones as a subset of type-2 clones. That restricts parameterization to consistent renaming of identifiers

with its container type and its element type and the loop body are clone parameters. The loop bodys X and Y may contain any code and can be of any size.

In the following chapters we present a specialized code clone detection algorithm that detects such parameterized clones. For that algorithm a data model is introduced to store intermediate results that are required for subsequent detection steps and to provide information about the detected clones in the end.

```
std:list<double> containerX;
for( std::list<double>::iterator
     iterX = containerX.begin();
     iterX != containerX.end();
     ++iterX )
{
   <loop-body-X>
}

std:vector<int> containerY;
for( std::vector<int>::iterator
     iterY = containerY.begin();
     iterY != containerY.end();
     ++iterY )
{
   <loop-body-Y>
}
```

Fig. 2. C++ example of a parameterized clone

The goal of our new detection algorithm is to find code idioms like the foreach functionality shown above and hopefully even larger idiomatic fragments. Such idioms are the basic blocks of an implementation of a software system. These idioms have the advantage to be at a higher level of abstraction than the elementary concepts provided by the programming language. The detection technique makes the idioms visible so that the programmers get aware of them. This simplifies and speeds up all tasks that require a deep understanding of program code especially during the maintenance phase.

The detection algorithm provides many clone groups for a given system. All of these groups possibly represent idioms. As an indication for which of these idioms are really interesting the algorithm also measures the number of instances for each clone group. If a clone group has many instances this is a hint for an idiom that is frequently used.

II. The Clone Representation Model

During the clone detection process a data model is populated to represent the gathered clone information. This clone representation model plays a central role in the detection algorithm. The clone fragments are added to the model one by one and incrementally build up a clone group hierarchy.

A. Clone group hierarchy

A clone group is a set of code fragments where the clone pair relation holds for each pair in this set:

$$\forall\, x, y \in \text{cloneGroup} : \text{clonePair}(x, y) \ . \qquad (1)$$

The term clone group is similar to the term clone class defined in [1]. The difference is that a clone class encompasses the elements of a clone group and transitively the elements of all subgroups of that clone group:

$$\text{cloneClass}(\text{cloneGroup}) = \text{cloneGroup} \cup \bigcup_{\text{subgroup} \in \text{subgroups}(\text{cloneGroup})} \text{cloneClass}(\text{subgroup}) \ . \qquad (2)$$

This is analog to the object-oriented terminology in Ada where a class encompasses a tagged type and all types derived from it. Subgrouping a clone group means that a subgroup represents clones of a larger extent than the supergroup, and the instances of the subgroup could also be instances of the supergroup. Figure 3 shows three C code fragments as an example and Fig. 4 the resulting clone group hierarchy: Part A of fragment 2 is a clone of part A of fragment 1. Part B of fragment 2 and part A of fragment 3 are clones of part B of fragment 1. And part B of fragment 3 is a clone of part C of fragment 2. Notice that the identifiers in fragment 3 are consistently renamed.

Fragment 1:

```
// A
for( int i=1; i<n; i++ ) {
   sum = sum + i;
}
// B
if( sum<0 ) {
   sum = n - sum;
}
```

Fragment 2:

```
// A
for( int i=1; i<n; i++ ) {
   sum = sum + i;
}
// B
if( sum<0 ) {
   sum = n - sum;
}
// C
while( sum<n ) {
   sum = n / sum;
}
```

Fragment 3:

```
// A
if( result<0 ) {
   result = m - result;
}
// B
while( result<m ) {
   result = m / result;
}
```

Fig. 3. Example C code fragments (taken from [1])

Parts A and B of fragment 1 together with parts A and B of fragment 2 form a clone group (CloneGroup 2 in Fig. 4).

Similary parts B and C of fragment 2 together with parts A and B of fragment 3 form CloneGroup 3. Both clone groups have two instances. Furthermore there is another clone group built up from part B of fragment 1, part B of fragment 2, and part A of fragment 3 (CloneGroup 1, encompassing three instances). This group is based on a smaller clone fragment that is also contained in the clone fragments of the other two clone groups. This relationship is modeled by saying CloneGroup 1 is a supergroup of CloneGroup 2 and CloneGroup 3 and CloneGroup 2 and CloneGroup 3 are subgroups of CloneGroup 1 (indicated by the arrows between the boxes).

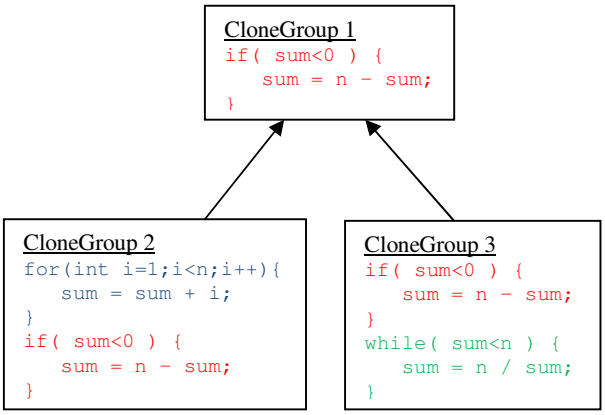

Fig. 4. Clone group hierarchy resulting from the code fragments in Fig. 3

The clone group hierarchy is a DAG (directed acylic graph), not just a tree. A subgroup can have multiple supergroups. This means that the clone fragment of the subgroup encompasses and extends the clone fragments of all supergroups. The extension of multiple clone groups is comparable with multiple inheritance in object orientation.

Notice that the code line `sum = n - sum;` does not form a clone group of its own although it is a clone with three instances. The reason is that this codeline is part of a bigger clone fragment which is represented by CloneGroup 1. There is no duplication of this line in the given code that does not also match CloneGroup 1. So the single line is not of any interest as a separate clone group here. This is important to avoid an explosion of the number of detected clone groups. The detection algorithm takes this into account by calculating the maximum extent of each clone fragment before inserting it into the clone representation model.

Similarly part A of fragment 1 and part A of fragment 2 do not form a clone group of their own because they can both be extended to the larger clone fragment represented by CloneGroup 2. On the other hand CloneGroup 1 is important although its clone fragment is also represented as part of CloneGroup 2 and CloneGroup 3. CloneGroup 1 is needed to express the commonalities of CloneGroup 2 and CloneGroup 3. If CloneGroup 1 would not be in the model it would not be obvious that there are similarities between CloneGroup 2 and CloneGroup 3. This is also essential for our detection algorithm (see chapter III.). Furthermore the number of instances of CloneGroup 1 is three. This smaller clone fragment has more instances than the subgroups both of which have only two instances. This might be relevant for the investigation of code idioms as described in chapter I.

B. Prototyping

For each clone group it is important to know what clone fragment it represents. Especially the extent of a clone fragment has to be stored to distinguish the corresponding clone group from its subgroups and supergroups that represent larger or smaller clone fragments. Our clone representation model realizes this by associating a prototype of the clone fragment to each clone group. A fragment prototype is a deep copy of a subtree of the given AST. It is copied from a clone group instance that is detected in the AST. This is one of the main steps required for the insertion into the data model (see chapter III.). Through storing the clone group prototypes as deep copies the clone representation model gets independent from the original AST. Although references to the original AST nodes are stored in the clone groups for each clone instance these references are not necessary for the clone representation model itself. They are just held to provide traceability from the clone groups to the origins in the source code. The clone representation model can be persisted without the original AST and the original source code. It is even possible to reconstruct an equivalent AST from that data model. The only exception is that the order of child nodes may differ from the original AST, e.g., the order of functions and procedures. But this does not affect the semantics of the reconstructed code.

A clone fragment prototype is referenced by a link to its root node. This is depicted in Fig. 5 as a composition relationship named prototypeRoot. The extent of the whole prototype subtree that is spanned by the referenced root node is denoted by the reflexive composition relationship from Node to itself. In general the original subtrees in the AST are nested more deeply. This is exactly how parameterized clones are expressed in this model. The details at the bottom of the original subtrees that are left out in the prototype are the clone parameters. On the one hand a clone group prototype establishes the upper horizontal cut for a group of parameterized clones through its root node. And on the other hand it draws a borderline between the common part of the clones and their clone parameters as the lower horizontal cut (see Fig. 1).

To simplify the implementation of the first version of this new clone detection algorithm it has been decided to associate a complete prototype to each clone group. This causes redundancy as the prototype of a subgroup redundantly encompasses all nodes that form the prototypes of its supergroups.

C. Measuring clone frequencies

In each clone group references to the root nodes of the clone group instances are stored. This implicitly expresses the size of the clone group as it is identical with the number of the stored references. The derived attribute count shown in Fig. 5 provides the number of clone group instances. The set of instance references and the count attribute encompass the instances that are included in the extended fragments of the

subgroups. E.g., count=3 for CloneGroup 1 in the example above although the three instances of CloneGroup 1 are all included in the instances of CloneGroup 2 and CloneGroup 3. As explained in the previous section it is not necessary to keep the instance references in a persistent storage of the clone representation model.

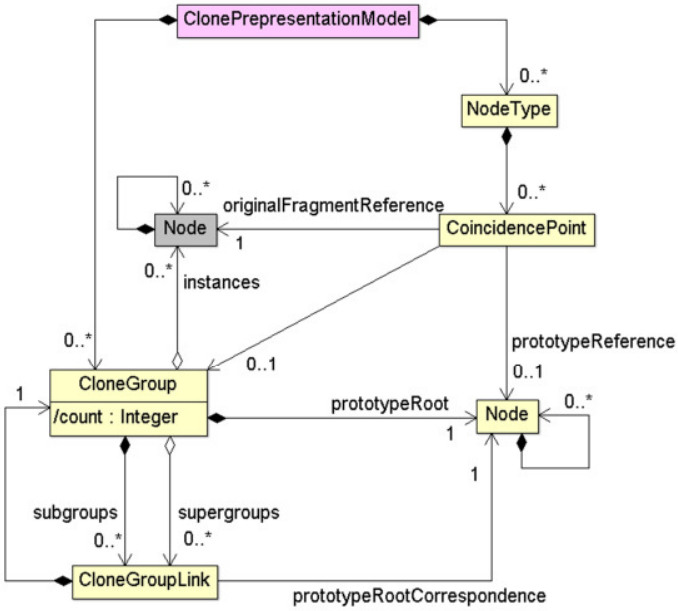

Fig. 5. The clone representation model as an UML class diagram

D. Details of the data model

The clone groups and their associated prototypes are the most important aspects of the clone representation model. But there are some further mostly technical details. Figure 5 depicts the complete schema of the model.

The links that interconnect the clone groups to form the clone group hierarchy are realized as separate objects of the type CloneGroupLink. This is necessary to establish the correspondences between the clone group prototypes. If a supergroup is extended by a subgroup the CloneGroupLink indicates which part of the subgroup prototype corresponds to the prototype of the supergroup. The reference prototypeRoot-Correspondence of a link object points to the node of the subgroup prototype that corresponds to the root node of the supergroup prototype. There are always links from a supergroup to a subgroup and from a subgroup back to a supergroup in order to make navigations in both directions possible. For our detection algorithm the links from the supergroups to the subgroups are important because it navigates that direction to search the right position for the insertion of the next clone fragment (see chapter III.). Notice that there may be multiple links between two clone groups. This occurs when the same supergroup prototype is contained in a subgroup prototype more than once at different positions. Then a separate link is needed for each correspondence position.

Specific for our detection algorithm are objects of the types NodeType and CoincidencePoint. A node type reflects of which kind an AST node is. There are different kinds of nodes for loops, for assignments, for variables, and so on. The corresponding data type NodeType is used for an initial clustering of the AST nodes based on their node type. A clustered AST node is represented by a CoincidencePoint. The CoincidencePoints have two purposes. The set of CoincidencePoints associated to a NodeType is processed one by one for each NodeType to construct the clone group hierarchy. A CoincidencePoint establishes a correspondence between an original AST node and the clone group prototypes. It has three outgoing references: The reference originalFragementReference points to the root node of the original AST subtree. This root node is of the NodeType the CoincidencePoint is associated to. The reference to CloneGroup indicates the clone group with the smallest prototype that contains a node that corresponds to the original AST node. And prototypeReference points to the corresponding node in that prototype.

There are two boxes for Node in the diagram. The grayed box indicates that the Node objects referenced here are original AST nodes whereas the other box represents the nodes of the prototypes.

III. The Clone Detection Algorithm

The basic idea of our clone detection algorithm is that two code fragments forming a clone pair have to coincide at least in one AST node. In our approach the coincidence of two AST nodes is just based on the node types, i.e., if both nodes are of the same node type their coincidence is proven. For each pair of nodes of the same node type the adjacent AST nodes are matched. This matching process starts with the two given nodes as we know that they coincide. That is the reason why we call them coincidence points. Then the matching process goes on with the direct and indirect child nodes and parent nodes until the maximum extension of the coincident subtrees is reached. These subtrees are the clone fragments we are searching. The approach to start matching by pointing to somewhere in the middle of the AST fits to our goal to detect parameterized clones as their instances are characterized to be inner subtrees of the AST.

A. The main algorithm

Our clone detection algorithm essentially relies on the clone representation model presented in chapter II. It works on an instance of that model which encompasses a set of clone groups and a set of node types (see Fig. 5). Initially both sets are empty. The first step of the algorithm is to visit all nodes of the given AST in a depth first tree walk and to cluster the nodes based on their node type. During this step NodeType objects are created on demand for the node types that occur in the tree walk. For each original AST node a CoincidencePoint object is created and associated to the NodeType object that represents its node type. The CoincidencePoint object is initialized with a reference to the original AST node (originalFragment-Reference). The reference from CoincidencePoint to Clone-Group and prototypeReference are null references initially.

The second step is to create an initial clone group. This group encompasses the whole source program as its clone fragment. The prototype associated to this clone group is a deep copy of the whole AST. During copying the AST all CoincedencePoint objects are further initialized with a reference to the initial clone group and a reference to the copied prototype node. The initial clone group is always the group with the largest clone fragment. All other clone groups created later on are direct or indirect supergroups of the initial clone group.

```
CloneRepresentationModel model;

void calculateCloneGroupHierachy()
{
   // 1. AST nodes clustering
   foreach node in AST
      model.nodeTypes[ node.type ].add( node );

   // 2. create the initial clone group
   CloneGroup initialGroup = new CloneGroup;
   initialGroup.prototypeRoot =
      deepCopy( AST.root);
   initialGroup.instances.add( AST.root );
   model.cloneGroups.add( initialGroup );

   // 3. insertion of the clone fragments
   foreach nodeType in model.nodeTypes
      foreach node in nodeType
         insertFragment( node,
            subset of nodeType with the
            nodes processed before node );
}
```

Fig. 6. Pseudocode of the main procedure of the detection algorithm

After the initialization, the clone group hierarchy is calculated. In this third step code clone fragments are successively added to the clone representation model. A clone fragment is either assigned to an already existing clone group or a new clone group is created and inserted into the clone group hierarchy. The main procedure (see Fig. 6) processes all clone fragment candidates. These candidates are determined by the original AST nodes represented by the CoincidencePoint objects as explained above. The CoincidencePoints are processed separately for each node type cluster. This reduces the number of required comparisons because a node can never coincide with a node of another node type.

B. Flooding

As described in section II.A. an important requirement on our detection algorithm is to create clone groups only for clone fragments of a maximum extension. This requirement is fulfilled by an approach we call flooding. The idea is similar to the flood fill functionality most paint programs provide. The flooding starts at a particular position and floods a region until a borderline is reached. In our algorithm flooding starts at a coincidence point and at the prototype of a clone group with a small extension simultaneously. The algorithm tries to flood an AST subtree around the coincidence point that matches the prototype of the clone group. The borderline is determined by the extension of the prototype and the restriction that the flooded nodes must coincide with the corresponding prototype nodes.

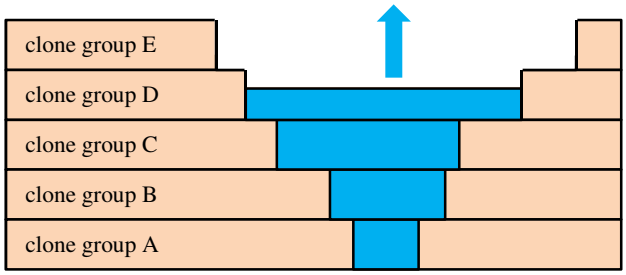

Fig. 7. The principle of the stepwise flooding

If the flooded area is equivalent to the whole clone group prototype the flooding continues with the subgroups of that clone group until a clone group is reached that cannot be flooded completely. This is the flooding level where the clone fragment is inserted. Figure 7 illustrates the principle of stepwise flooding that is used is used here.

C. The insertion of a clone fragment

To illustrate how a clone fragment is inserted into the clone representation model Fig. 8 shows simplified pseudocode of the function insertFragment that is called from the main procedure calculateCloneGroupHierachy. The function insertFragment expects a reference to the root node of a clone fragment candidate as its first parameter. We call it a candidate because it may turn out during the following process that no insertion is needed for that code fragment. The second parameter is a set of CoincidencePoints. These CoincidencePoints are possible start points of the matching process. The insertion initiated by calculateCloneGroupHierachy starts with the CoincidencePoints of the corresponding cluster gained from the clustering step. This ensures that it is not tried to match a subtree that does not coincide at all.

After the second initialization step all CoincidencePoints point into the initial clone group that encompasses a copy of the whole AST. At the beginning the insertion process tries to match clone fragment candidates with subtrees of the prototype of that initial clone group. When such a matching is found a new clone group is created and inserted above the initial clone group as its supergroup. The coinciding subtree is associated to the new clone group as its prototype. The two matching clone fragments are stored as its instances. Another important step is that both CoincidencePoints that have been the start points of the match are modified to point to the new clone group and into its prototype. The further detection process relies on the precondition that the CoincidencePoint objects always point to that clone group with the smallest prototype where the corresponding AST node appears first. Because now two CoincidencePoints reference the same prototype node one of them is marked to be skipped in the further process.

For the insertion of the next clone fragment either a link from a CoincidencePoint that still points to the initial clone

group is followed or the matching process steps to the new supergroup. The first case is a similar to the previous insertion. In the second case a new clone group might be created above the two other clone groups or in between of them. Again the references in the CoincidencePoint objects must be updated. This update guarantees that the matching process always starts at the smallest possible prototypes. The process proceeds successively to bigger prototypes via the subgroup links until the right insertion position is found as described in the previous section.

```
enum MatchResult { NO_MATCH, BIGGER, SMALLER };

bool insertFragment
( CoincidencePoint fragment,
  inout list<CoincidencePoint> startPoints )
{
   Node fragmentRoot =
      fragment.originalFragementReference;
   foreach startPoint in startPoints
   {
      CloneGroup group = startPoint.cloneGroup;
      switch( match( fragmentRoot,
                     group.prototype ) )
      {
      case BIGGER:
         if( not insertFragment( fragment,
                                 group.subgroups ) )
            addInstance(fragment, startPoint);
         return true;

      case SMALLER:
         CloneGroup newGroup
                        = new CloneGroup;
         newGroup.prototype
            = createPrototype( fragmentRoot,
                               group.prototype );
         CloneGroupLink link =
            new CloneGroupLink( newGroup );
         addInstance( fragment, link );
         newGroup.subgroups.add( startPoint );
         newGroup.subgroups.add( fragment );
         startPoints.replace(startPoint, link);
         return true;

      case NO_MATCH:
         break;
      }
   }

   // no matching prototype found
   return false;
}
```

Fig. 8. Simplified pseudocode that shows how further code fragments are inserted into the clone representation model

To realize all that there is a distinction of cases in the insertFragment function. The possible match start points are processed one by one. For each start point it is tried to match the corresponding prototype. If the whole prototype coincides with the clone fragment candidate at hand flooding proceeds to the subgroups by a recursive call of insertFragment. In this case the result of the matching is BIGGER because the clone fragment is bigger than the prototype it has been matched with. If the prototype cannot be completely matched the function match provides the result SMALLER. In this case a new clone group is created between the current group and the supergroup processed before. The prototype of the new clone group is provided by the function createPrototype. It creates a deep copy of the maximum coincidence region that results from a match starting at nodes passed as function parameters. The links between the clone groups are updated. The new clone group initially has two subgroups. One is the group the new one has been added above and the other is the group the just added clone fragment has referenced before (usually the initial clone group). Several details are left out in Fig. 8, especially some conversions and the creation of reverse links.

If the prototype and the clone fragment candidate coincide only in their root nodes the result of the match function is NO_MATCH. Clone fragments that encompass just one node are irrelevant. We do not create clone groups for such clone fragments. The size of a clone group prototype is at least two nodes.

Notice that only one of the start points processed in insertFragment can have a BIGGER or SMALLER match with the clone fragment candidate. For all the others the result is NO_MATCH. This is guaranteed by the way the clone representation model is constructed. If there would be multiple start points with the match result BIGGER or SMALLER the commonalities had been extracted to a new supergroup that had replaced these start points in the start points list.

D. Fragment matching and prototyping

The function match called by insertFragment starts with pointing somewhere into the original AST and pointing into a clone group prototype. As illustrated in Fig. 9 the matching process includes three steps. The first step checks if the two nodes pointed to at the beginning coincide. If they do not coincide the match result is NO_MATCH. In the second step the matching walks downwards through the subtrees that are spanned by the start points. The subtrees are processed simultaneously in preorder. If the comparison of two corresponding inner nodes fails it is shown that the coincidence does not encompass the whole extent of the given prototype. In the case of a successful comparison the matching process proceeds to the child nodes. If the prototype has a child node that does not exist at all in the code it is compared with the coincidence is also incomplete. As the start points are somewhere in the middle of the trees that shall be compared it is necessary to walk upwards from the start point as well. This is done in the third matching step. It walks to the parent nodes of the start nodes, compares them, and processes the subtrees spanned by their child nodes in the same way as described for the second step. Then it proceeds to the further parent nodes until the root node of the prototype is reached. Similar to the second step, during this step it can be found that the coincidence with the prototype is not complete. If coincidence does not encompasses more than one node at the end the match result is

NO_MATCH. Otherwise it is BIGGER for a complete coincidence and SMALLER for an incomplete one. Notice that the only case in which the compared trees are equal is a comparison with the prototype of the initial clone group starting at the same node. As this is never done in our algorithm this case is not considered in the matching process.

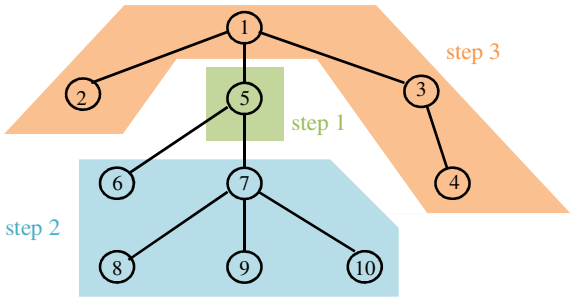

Fig. 9. Example of the matching process with node 5 as start point

The process for creating a clone group prototype is similar to the matching process. The nodes of the compared subtrees are visited in the same way. The difference is that the corresponding nodes are not only compared but also copied if they coincide. If there is a coincidence between two nodes a new node of the same node type is created. To connect the new nodes to a tree they are linked in the same way as the nodes in the original trees.

E. Runtime complexity

The three steps of the main procedure calculateCloneGroupHierachy are processed successively (see Fig. 6). To determine the overall runtime complexity class the step with the highest complexity is decisive. The first step visits each AST node once. This implies a runtime complexity of $\dot{O}(n)$ for that step with n as the number of AST nodes. The deep copy in the second step copies each AST node once. So this step is in $\dot{O}(n)$ as well. The inner loop body in step three is passed once for each AST node as the node sets associated to the node types are all disjoint. But here the internals of the called function insertFragment have to be considered. To search for the right position for the insertion the algorithm walks down a path through the clone group hierarchy. In the worst case this path encompasses all clone groups and the upper bound of the number of clone groups is n because there is at most one new clone group created during the processing of an original AST node. For the matching between a clone fragment candidate and a clone group prototype up to n nodes have to be compared as the size of a prototype is limited by the size of the AST. The possibly following creation of a new prototype also processes up to n nodes. The overall runtime complexity is in $\dot{O}(n^3)$. Baxter et al. have calculated a complexity of $\dot{O}(n^3)$ for a simple AST based clone detection approach [2]. Our algorithm is in the same runtime complexity class as a simple approach for detecting type-2 clones. But our algorithm detects clones of a more general clone type encompassing parameterized clones.

IV. Algorithm Implementation

Currently the code clone detection algorithm described here is partly implemented based on Bauhaus. Bauhaus is a program analysis and reengineering framework [6]. As it provides a C/C++ frontend C and C++ source code can be examined by the code clone detection algorithm. The detection algorithm itself is written in Ada. Bauhaus translates the incoming source code to an intermediate representation called IML. This intermediate representation is a graph that contains the AST and is the input to our AST based algorithm.

The Bauhaus IML is based on a specification that defines classes for all AST node types and their relationships. The classes required for the clone representation model as shown in Fig. 5 have been added to that specification. A generator produces Ada code for all classes in the specification, including data type definitions, access methods, and iterator functionalities.

To realize the clone fragment matching and the creation of prototypes (see section III.D.), Ada classes have been implemented following the visitor design pattern. This is necessary because AST nodes of different node types have to be handled differently.

V. Related Work

Several approaches for AST based clone detection techniques have been published. One of the publications that influenced the clone detection community a lot has been presented by Baxter et al. [2]. Their algorithm starts with hashing all subtrees in a given AST. This is similar to our clustering of the AST nodes. The difference is that Baxter et al. calculate a hash value from the whole subtree that is spanned by a root node because they are only interested in adhesive code blocks as clone fragments but not in clones from the middle of the AST as we are. The second step in the Baxter et al. algorithm is to find clone pairs. If a larger clone pair is found all similar clone pairs that are smaller are thrown away. The idea was to reduce the amount of data that is provided as the algorithm result. They preferred large fragments to smaller fragments. In contrast we are also interested in the frequencies of clone occurrences. The problem of the possibly huge amount of result data is handled in two ways in our solution: The result is structured with a classification in clone groups. And not all possible clones are added to the clone representation model. The fragment flooding assures that only the fragments of maximum extend are taken into account. A calculation of clone groups or clone classes is not included in the Baxter et al. algorithm. Further AST based clone detection approaches are presented in [7] and [8].

Branda S. Baker has established the idea of suffix trees in her token-based clone detection approach [3]. You can view our clone representation model as a generalization of the suffix tree idea. Although our approach is not token stream based but AST based there is a similarity between token stream suffixes and subtrees in an AST. Similar to our clone representation model a suffix tree provides clone groups and inclusion relationships between them because inner nodes that are deeper in the suffix tree represent larger clone fragments beginning

with the same prefix as the other suffix tree nodes on the path to the suffix tree root. This results in inclusion relationships that enlarge clone fragments at their end. In contrast our clone representation model allows expressing enlargements at any branch of a subtree and even at the top of a subtree. This is especially important for adding further clone fragments to the model incrementally.

VI. Further Research

The focus of this paper is to show the feasibility of an algorithm that detects parameterized clones. There are promising possibilities to improve the performance of the algorithm. An optimal solution could avoid the redundancy mentioned in section II.B. by just storing the additional prototype nodes for subgroups. As a result for each original AST node at most one corresponding prototype node would be required in the total clone representation model. The memory amount would the reduced from $O(n^2)$ to $O(n)$ with n as the number of nodes in the original AST. The suggested optimization could reduce the runtime complexity from $O(n^3)$ to $O(n^2)$. If the clone groups would hold just those prototype nodes that are additional to their supergroup prototypes only up to n node comparisons are required for the whole matching process for a clone fragment candidate. The effort for handling one clone fragment candidate would be reduced from quadratic to linear complexity. Furthermore it is an open question if the number of clone groups really increases linearly with the size of the AST. We expect that the number of clone groups ascends slower than $O(n)$ in practice. The algorithm has to be applied on some real software systems to clarify that.

The matching process could be further improved in the handling of sequences of nodes. This is especially important for statement sequences. Our current implementation allows detecting a subgroup of a clone group if it adds nodes at the end of a node sequence. It would be more flexible if additions could be located at any position in a node sequence.

Currently our algorithm is based on the AST of the source code of a given software system. The same detection approach could also be applied on a program dependency graph (PDG) derived from the AST. This makes it possible to view clone fragments as equivalent although they may differ in the order of their statements. To free the clone detection even further we want to add some transformation rules that make it possible to find type-4 clones [1].

VII. Conclusion

Our new code clone detection algorithm shows that it is possible to calculate all parameterized clones that are contained in a given software system in $O(n^3)$ with n as the size of the given AST.

Beside the detection algorithm a clone representation model has been developed. With that model the detected parameterized clones, their clustering into clone groups, and the relationships between the clone groups forming a clone group hierarchy are expressed explicitly. The model encompasses clones of all granularities simultaneously.

Our detection algorithm relies on that clone representation model in two ways. The detection results are provided in the form of this model and the model is also used to manage all intermediate results during the detection process. The clone representation model could also be used in combination with any other clone detection tool. For that the model must be calculated in a post processing step from the set of clone pairs that is provided by most clone detection tools. But in contrast to other clone detection tools our detection algorithm is specifically designed to populate the model with parameterized clones.

The parameterized clones detected by our algorithm and provided in the clone representation model are candidates for code idioms. To support the decision if a clone is really a relevant code idiom the frequency of the occurrences is calculated for each clone as well. The final selection of relevant idioms should be checked manually because the relevance is usually based not only on the occurrence frequency.

References

[1] Chanchal Kumar Roy and James R. Cordy, “A survey on software clone detection research”, technical report, Queen’s University, Canada, 2007.

[2] Ira Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant Anna, and Lorraine Bier, “Clone Detection Using Abstract Syntax Trees”, in Proceedings of the 14th International Conference on Software Maintenance (ICSM’98), pp. 368-377, Bethesda, Maryland, November 1998.

[3] Brenda S. Baker, “On Finding Duplication and Near-Duplication in Large Software Systems”, in Proceedings of the Second Working Conference on Reverse Engineering (WCRE’95), pp. 86-95, Toronto, Ontario, Canada, July 1995.

[4] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer, “A Language Independent Approach for Detecting Duplicated Code”, in Proceedings of the 15th International Conference on Software Maintenance (ICSM’99), pp. 109-118, Oxford, England, September 1999.

[5] Torsten Görg, “A Model-Based Approach to Type-3 Clone Elimination”, in Proceedings of the 14. Workshop Software-Reengineering (WSR 2012) of the Gesellschaft für Informatik (GI) special interest group Software-Reengineering, pp. 21-22, Bad-Honnef, May 2012.

[6] Aoun Raza, Gunther Vogel, and Erhard Plödereder, “Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering”, in Proceedings of Ada Europe 2006, LNCS 4006, pp. 71-82.

[7] Vera Wahler, Dietmar Seipel, Jürgen Wolff v. Gudenberg, and Gregor Fischer, “Clone detection in source code by frequent itemset techniques”, in Proceedings of Fourth IEEE International Workshop on Source Code Analysis and Manipulation, pp. 128-135, September 2004.

[8] William S. Evans, Christopher W. Fraser, and Fei Ma, “Clone Detection via Structural Abstraction”, in Proceedings of 14th Working Conference on Reverse Engineering (WCRE 2007), pp. 150-159, October 2007.