Literature Survey of Software Clones

Vishwachi
Research Scholar,  
AKGEC, Ghaziabad, India

Sonam Gupta  
Assistant Professor,  
Department of Computer Science and Engineering, AKGEC, Ghaziabad, India

ABSTRACT
Code cloning is the procedure where the developers reuse the code fragments implementing the paste option. They may or may not make the modification in the source code. The code thus developed after copying is known as clone. It is the synonym of duplicate. In the year 2002, Ira Baxter coined the term clones as the segments of code that are similar according to some definition of similarity. The similarity can be based on text, syntactic or semantic. Studies have revealed that almost 10-15% of the source code in large software are part of single or more clones[1]. Clones have adverse impact on the software maintenance, thus identification of clones is beneficial. In the past decade many tools have been developed to detect the clones but none was able to correctly identify all types of clones. In this paper the literature survey of all the clone detection techniques has been done. Along with this it also propose an approach which will use a combination of tree and token based approach in order to detect the code clones.

Keywords
Clones, textual comparison, LWH approach, token based approach, PDG approach, metric comparison, AST approach.

1. INTRODUCTION
A code clone is a section of code in the source files which is similar or identical to another code section[4]. The similarity can be based on text, syntax or semantic (behavioral). Code clone generally results from the practice of copy and paste followed by the developers. They result in increased maintenance efforts. Thus various tools have been developed to detect the code clones. Automated code clone techniques and tools utilize different similarity measures to find clones in code[6]. Various techniques available for detecting the code clones are textual comparison, metric based approach, token based approach, AST based approach, PDG approach, LWH approach, CRD approach. In this paper we will briefly describe the clones and the measures to identify them. The consequences of cloning are that cloning of code generally increases the maintenance effort. If the code is redundant then changes must be made consistently multiple times. This effort could have completely avoided if the code would have been implemented only once in a function. Code cloning is a purposeful implementation strategy which may make sense under certain circumstances [3]. Generic solutions can become overly complicated.

The paper is organized as: section II contains types of clones, section III describes the procedure of clone detection, section IV gives the detailed techniques of clone detection along with the limitations and advantages of each. And finally section V gives the proposed approach and section VI includes the conclusion.

2. TYPES OF CLONES
Clones are generally divided into four main categories:
Type 1: program fragments which are identical copies of each other expect for the whitespaces and comments variations.

Example:

```c
int i=1;  /*initializing the value of i */
if (i<=5) printf(“Continue…..”);
else printf(“skip..”);
i++;
// fragment 1
```

Type 2: Program fragments which are syntactically identical copies; except some changes in variables names, data type, identifier name, etc.

Example:

```c
float k=1; /*initialize the value of i */
if(k<=10) printf (“Hello…..”);
else printf (“Bye…..”);
k++;
//fragment 2
```

Type 3: is a copied fragment with further modifications. Statements can be changed ,added or removed in addition to variations in identifiers, literals, types, layout and comments.

Example:

```c
int i=1; /* initializing value of i */
if(i<=10) printf(“Hello…..”);
else printf(“Bye…..”);
i++;
//fragment 1
```

Type 4: program fragments which are semantically different copies; except some changes in variables names, data type, identifier name, etc.

Example:

```c
int i=1; /* initializing value of i */
if (i<=5) printf (“Continue…..”);
else printf(“skip..”);
i++;
// fragment 2
```
3. Match Detection
The comparison units take transformed code as input and compares the transformed comparison units to each other to discover matches [4].

4. Formatting
Here the source coordinates of each clone pair obtained in the comparison phase are mapped to their positions in the original source files.

5. Filtering
Here the clones are manually analyzed, filtered and ranked or they may be fed under automated heuristics.

6. Aggregation
Clones may be aggregated to clone classes with the aim to diminish the measure of data or gather overview statics.

3. CLONE DETECTION PROCESS
The detection of clones has now become one of the most important part of the development process as the clones create trouble later in the maintenance phase. The process of clone detection can be sub-divided into six phases. They are:

1. Preprocessing
The detection of code cloning starts with the partitioning of the source code. This phase is mainly responsible for:

   • Removal of all the comments, white spaces etc.
   • Determine source units: the source code obtained after the removal of white spaces and comments is then partitioned into a set of disjoint pieces known as source units which are the largest source sections suspected to be involved in direct clone relations with each other[4].
   • Determining the granularity: source units may further be partitioned into smaller units say into lines or tokens for a comparison purpose.

2. Transformation
After preprocessing step, the source code of the comparison units is transformed to a proper intermediate format for comparison [4]. Transformation includes extraction and normalization. Extraction may involve tokenization, parsing, control and data flow analysis depending on the approach we are following in the detection process. Normalization is an optional step.

4. APPROACHES OF CLONE DETECTION

4.1 Textual Comparison
This approach compares whole lines to each other textually. The targeted source code is considered to be the sequence of the strings or lines in order to find the match/clone the two code parts are compare to each other. If the two parts of the code found out to be similar then they are considered as clones. The approach followed in textual comparison is light weighted and are able to detect the clones accurately with higher recall values, where recall refers to the overall percentage of clone exist in the source code that have been detected by the clone detector [1]. The work done in this technique has been summarized in table 1.

Example:

Type 4: Two or more code fragments that perform the same computation, but implemented through different syntactic variations.

Example:

<table>
<thead>
<tr>
<th>Preprocessing</th>
<th>Transformation</th>
<th>Match Detection</th>
<th>Formatting</th>
<th>Filtering</th>
<th>Aggregation</th>
</tr>
</thead>
</table>

Fig 1. Clone detection procedure
4.2 Token Based Approach

In token based clone detection techniques, firstly, tokens are extracted from the source code by lexical analysis. Then sequence is formed from some set of tokens which are then compared in order to find the clone. They are fast with high recall values. The most popular token-based tool named CCFinder was developed by T.Kamiya and S.Kusumoto [19] in the year 2002. Since then, it is a popular tool among researchers and has been used widely for code clone analysis and management. Researchers are working to enhance the output of CCFinder for example Basit et al. [20] used CCFinder to study the patterns of clones in a standard template library. They increased the threshold of CCFinder to detect the smaller clones too. Another tool based on token approach is CP-Miner which uses frequent item set mining in order to detect the bugs in the softwares produced due to clones. Yamashina et al. [42] designed a tool called CCFinderX. The work done in this technique has been summarized in table 2.

Table 1. Work done on text based approach

<table>
<thead>
<tr>
<th>Sn. no.</th>
<th>Author</th>
<th>Tool</th>
<th>Year</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Wette l et al. [17]</td>
<td>Dude</td>
<td>200 5</td>
<td>Can detect duplicating chains consisting of smaller size exact clones.</td>
<td>Can not be applied to large systems.</td>
</tr>
<tr>
<td>2.</td>
<td>C.K. Roy et al. [30,31]</td>
<td>NICA D</td>
<td>200 8-200 9</td>
<td>Can detect type 3 clones very effectively as compared to other text based tools.</td>
<td>Not exactly text-based but rather hybrid as it exploits the benefits of tree-based structural analysis.</td>
</tr>
<tr>
<td>3.</td>
<td>S. Lee et al. [18]</td>
<td>SDD</td>
<td>200 5</td>
<td>Capable of detecting clones in large sized systems.</td>
<td>Its accuracy is not high.</td>
</tr>
<tr>
<td>4.</td>
<td>Baker et al. [36,37]</td>
<td>Dup</td>
<td>199 2-199 9</td>
<td>Can detect clone even if the names of variables are different.</td>
<td>Uses large search space as hashing is to be applied. It cannot detect clones if source code is written in different styles.</td>
</tr>
</tbody>
</table>

| 5.      | Cordy et al. [39] | 200 4 | Capable of detecting near-miss clones. | Precision value is not high. |
| 6.      | Ducasse et al. | The tool is language independent | Not able to detect meaningful clones |

4.3 Metric Based Approach

In this technique, we gather different metrics for code fragments and compare these metric vectors instead of comparing the code directly. In this approach metric values for different methods are calculated to extract the potential clone pairs. The metrics used may involve number of lines, number of arguments, number of function calls etc. The two methods whose metrics come out to be similar are considered as clone pairs. The major advantage of this technique is that it can detect both the syntactically and semantically similar clones. The work done in this technique has been summarized in table 3.

Table 2. Work done on token based approach

<table>
<thead>
<tr>
<th>Sno.</th>
<th>Author</th>
<th>Tool</th>
<th>Year</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>T.Kamiya et al. [19]</td>
<td>CCFinder</td>
<td>200 2</td>
<td>Till date the most popular tool used for clone analysis and management.</td>
<td>It is unable to detect smaller clones.</td>
</tr>
<tr>
<td>2.</td>
<td>Basit et al. [20]</td>
<td>200 5</td>
<td>Enhanced CCFinder by increasing its threshold to detect smaller clones too.</td>
<td>Unable to find out the semantic similarity between the codes.</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Z Li et al. [44]</td>
<td>CP-Miner</td>
<td>200 6</td>
<td>Can detect bugs in software induced due to cloning.</td>
<td>Precision is not very high.</td>
</tr>
<tr>
<td>4.</td>
<td>Yamashina et al. [42]</td>
<td>CCFinder X</td>
<td>200 9</td>
<td>Tokens are fed as input to suffix array which results in a fast retrieval.</td>
<td>Needs improvement in ranking algorithm.</td>
</tr>
<tr>
<td>5.</td>
<td>Sasaki et al.</td>
<td>FCFinder</td>
<td>201 0</td>
<td>Used hashing</td>
<td>It does not accept source files written in two or more programming language.</td>
</tr>
</tbody>
</table>
Table 3. Work done on Metric based approach

<table>
<thead>
<tr>
<th>Sno.</th>
<th>Author</th>
<th>Year</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Mayrand et al. [21]</td>
<td>1996</td>
<td>Was one of the first approach to compare metrics obtained from AST of source code.</td>
<td>Not able to identify segments which are based on copy-paste operation.</td>
</tr>
<tr>
<td>2.</td>
<td>Kontogiannis et al. [22]</td>
<td>2004</td>
<td>Applies dynamic programming on the lines of source code by using minimum edit distance between them. Thus, it is able to detect the similarity more precisely.</td>
<td>Not being able to find the exact clones. It can only find out the similarities between the codes.</td>
</tr>
<tr>
<td>3.</td>
<td>Perumal et al. [43]</td>
<td>2010</td>
<td>Used fingerprint technique to detect the clones.</td>
<td>The technique used is quite costly.</td>
</tr>
<tr>
<td>4.</td>
<td>Li and Sun [44]</td>
<td>2010</td>
<td>The technique used in this tool is scalable as well as accurate.</td>
<td>It is yet to be verified for different systems.</td>
</tr>
<tr>
<td>5.</td>
<td>Lovoie et al. [45]</td>
<td>2010</td>
<td>Technique used is based on graphics processing unit (GPU) which results in increased performance.</td>
<td></td>
</tr>
</tbody>
</table>

4.4 Program Dependency Graph (PDG) Approach

PDG is a semantic (behavioral) based approach. PDG considers the semantic information which is encoded in the form of a dependency graph that captures the data flow and control information. Clones may be identified as isomorphic sub graphs in a PDG [3]. R. Komondo et al. [24] developed a tool based on PDG approach which uses program slicing to find out the isomorphic subgraphs. Its main feature is that it helps in detecting the non-contiguous clones. Another tool Scorpio was developed by Higo and Kusumoto [25] in the year 2011 which applies two-way slicing to detect clones. It was developed with the aim to address the problem of slow detection of contiguous clones which the existing systems faced. Krinke [35] used PDG as an iterative approach for finding maximal similar sub-graph but it suffers from the shortcoming that it was not able to give a formula that can be used on any type of system to find the clone[35]. All the researchers using PDG technique came to the conclusion that although PDG-based techniques can find non-contiguous clones but it cannot be applied to large systems[35]. The major disadvantage of this approach is that sub graph comparison is quite costly in this approach. The algorithms which uses this technique returns the approximate results. The tabular representation of the work done so far using this technique is given in table 4:

Table 4. Work done on PDG based approach

<table>
<thead>
<tr>
<th>Sno.</th>
<th>Author</th>
<th>Year</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Horwitz et al.</td>
<td>1990</td>
<td>Can identify syntactic and semantic difference between two versions of program.</td>
<td>Cannot be applied to large systems.</td>
</tr>
<tr>
<td>2.</td>
<td>Krinke et al. [35]</td>
<td>2000</td>
<td>Finds maximal similar subgraphs</td>
<td>Not able to give a single formula which can be used on any type of system for finding clone.</td>
</tr>
<tr>
<td>4.</td>
<td>Higo et al. [25]</td>
<td>2011</td>
<td>Based on no. of PDG specializatio n for Java language and heuristics</td>
<td></td>
</tr>
</tbody>
</table>

4.5 Abstract Syntax Tree (AST) Approach

The most commonly used representations in order to transform the source code into tree structure is Abstract Syntax tree and parse trees. It can be used to find out the syntactic differences between the two parts of the same source code. This technique is generally based on grammar and a parse tree is generated for both the parts of the code. Detection of the clone is applied synchronously to both the trees and it is based on the LCS( longest common subsequence). If the subsequence of the two parts comes out to be similar then they will be considered as clones. The work done in this technique has been summarized in table 5.
Table 5. Work done on AST based approach

<table>
<thead>
<tr>
<th>Sn o.</th>
<th>Author</th>
<th>Year</th>
<th>Tool</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.</td>
<td>Jiang et al. [46]</td>
<td>2007</td>
<td>Deckard</td>
<td>Capable of finding the behavioral similarity.</td>
<td>The approach used is heavy-weighted.</td>
</tr>
<tr>
<td>3.</td>
<td>Falke et al. [29]</td>
<td>2008</td>
<td>Used syntax tree. Has advantage of precision of syntax tree and high speed of syntax tree.</td>
<td>Takes longer time to traverse the tree.</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>Ekoko et al. [49]</td>
<td>2008</td>
<td>Clone Tracker</td>
<td>Detect the clones in Java codes</td>
<td>Results in large number of false positives.</td>
</tr>
<tr>
<td>5.</td>
<td>W.S.Evans et al. [48]</td>
<td>2009</td>
<td>Asta</td>
<td>Works on structural abstraction of arbitrary subtrees of AST.</td>
<td></td>
</tr>
</tbody>
</table>

4.6 Hybrid Approach

As the name implies it is a hybrid technique so it will combine two or more of the above techniques. LWH (light weight hybrid) combines the textual comparison and metrics based approach in order to detect the method-level syntactic and as well as semantic clones. The tool developed using this technique accepts the source code and separates the functions/methods present in it. Also, it forms the template of each of the method present. After this, the code metrics is computed for each method and are stored in a database. The methods whose metrics values comes out to be nearly similar are subjected to textual comparison to detect the actual clone pairs. Egambaram Kodhai et al. [1] in the year 2014 developed a tool named CloneManager which used LWH approach. The tool is able to detect all four types of the clones with high precision. It first converts the source code into templates and then apply the metrics approach to find the similarities between different parts of the source code. Maeda [50] in the year 2009 introduced a technique based on PALEX source code representation. It is language independent and uses a suffix tree for comparison. Chilowicz et al. [51] used suffix array and metrics. The technique starts with collecting the tokens using lexical analysis. Basit et al. [52] developed a tool named Clone Miner in the year 2009 which used frequent item set mining and works on the output of a token based clone detection tool named RTF. Its disadvantage is that it doesn’t implies the refactoring method. The major advantage of this approach is that it is light weighted so the time it takes to detect the clones is quite low as compared to other techniques. The accuracy of the tool developed using this technique comes out to be 88-100% which is considerably high.

5. PROPOSED APPROACH

Till now, we have seen the advantages and disadvantages of all the well known approaches of locating the clones in source code. There is not a single approach which is capable of detecting all four kinds of clones precisely with 100% efficiency. The text based approach can detect the clones of only 2 types (in some cases upto type 4). So, here we propose a hybrid approach including the tree and token based approach. We will construct an AST of the given source code and will traverse it using DFS technique. After that, we will store the leaf nodes of the tree as tokens and the tokens of the same kind will be stored in the list. Then the Levenshtein distance between them will be find out. The two list of tokens among whom the Levenshtein distance comes out to be minimum than a particular threshold can be considered as clones.

6. CONCLUSION

In this paper, a review about the clones and different techniques to detect them is shown. Clone are harmful as they increase the maintenance efforts. So, it is better to detect them beforehand in order to avoid any kind of problems in future. Also, the refactoring procedure is not easy because of the cost and risk associated with refactoring. Therefore, one of the above mentioned approach may be used at the time of development of the source code before handing over the product to the client so that in maintenance phase the product doesn’t encounter much difficulty. The studies shows that the most advantageous method of detecting clones is the combination of metrics and textual approach as they results in light weight tool development occupying less space and resulting in a small execution time as compared to other techniques.

7. REFERENCES


[27] Tool SimScanhttp://www.blue-edge.bg/download.html
[28] Project Bauhaus http://www.bauhaus-struggart.de


