A Survey of Software Clone Detection Techniques

Abdullah Sheneamer  
Department of Computer Science  
University of Colorado at Colo. Springs, USA  
Colorado Springs, USA

Jugal Kalita  
Department of Computer Science  
University of Colorado at Colo. Springs, USA  
Colorado Springs, USA

ABSTRACT

If two fragments of source code are identical or similar to each other, they are called code clones. Code clones introduce difficulties in software maintenance and cause bug propagation. Software clones occur due to several reasons such as code reuse by copying pre-existing fragments, coding style, and repeated computation using duplicated functions with slight changes in variables or data structures used. If a code fragment is edited, it will have to be checked against all related code clones to see if they need to be modified as well. Removal, avoidance or refactoring of cloned code are other important issues in software maintenance. However, several research studies have demonstrated that removal or refactoring of cloned code is sometimes harmful. In this study, code clones, common types of clones, phases of clone detection, the state-of-the-art in code clone detection techniques and tools, and challenges faced by clone detection techniques are discussed.

Keywords

Software Clone, Code Clone, Duplicated Code Detection, Clone Detection

1. INTRODUCTION

When a programmer copies and pastes a fragment of code, possibly with minor or even extensive edits, it is called code cloning. Code clones introduce difficulties in software maintenance and lead to bug propagation. For example, if there are many identical or nearly duplicated or copy-pasted code fragments in a software system and a bug is found in one code clone, it has to be detected everywhere and fixed. The presence of duplicated but identical bugs in many locations within a piece of software increases the difficulty of software maintenance. Recent research [5, 6, 13, 11] with large software systems [6, 15] has detected that 22.2% of Linux code has clones. Kamiya et al. [14] has reported 29% cloned code in JDK. Baker [15] has detected clones in large systems in 13% - 20% of the source code. Baxter et al. [16] also have found that 12.7% code is cloned in a large software system. Mayrand et al. [19] have also reported that 5% - 20% code is cloned. Code clone detection can be useful for code simplification, code maintainability, plagiarism detection [14], copyright infringement detection, malicious software detection and detection of bug reports. Many code clone detection techniques have been proposed [1]. The focus of this paper is to present a review of such clone detection techniques.

The rest of the paper is organized as follows. Section 2 discusses background material. Prior survey papers on the topic of code clone detection are introduced in Section 3. In Section 4, the seven phases in clone detection are described. Several clone detection approaches, techniques and tools are discussed in great details in Section 5. Evaluation of clone detection techniques is discussed in Section 6. How detected clones can be removed automatically and areas related to clone detection are discussed in Section 7. Open problems related to code clone detection research are covered in Section 8. Finally, the paper is concluded in Section 9.

2. BACKGROUND

Code clones are used frequently because they can be created fast, and easily inserted with little expense [2]. However, code clones affect software maintenance, may introduce poor design, lead to wasted time in repeatedly understanding a fragment of poorly written code, increase system size, and reincarnate bugs that are present in the original of code segment. All these make it a difficult job to maintain a large system [1, 2].

Additional reasons that make code clone detection essential are the following. 1) Detecting cloned code may help detect malicious software [3]. 2) Code clone detection may find similar code and help detect plagiarism and copyright infringement [2, 4, 5]. 3) Code clone detection helps reduce the source code size by performing code compaction [6]. 4) Code clone detection also helps detect crosscutting concerns, which are aspects of a program that impact other issues that arise when code is duplicated all over the system [6].

2.1 Basic Definitions

Each paper in the literature defines clones in its own way [1]. Here, common definitions which used throughout this paper are provided.

Definition 1: Code Fragment. A code fragment (CF) is a part of the source code needed to run a program. It usually contains more than five statements that are considered interesting, but it may contain fewer than five statements. It can contain a function or a method, begin-end blocks or a sequence of statements.

Definition 2: Software Clone/Code Clone/Clone Pair. If a code fragment CF1 is similar to another code fragment CF2 syntactically or semantically, one is called a clone of the other. If there is a relation between two code fragments such that they are analogous or similar to each other, the two are called a clone pair.
**Definition 3: Clone Class.** A clone class is a set of clone pairs where each pair is related by the same relation between the two code fragments. A relation between two code fragments is an equivalence relation which is reflexive, symmetric and transitive, and holds between two code fragment if and only if they are the same sequence.

### 2.2 Types of Clones

There are two groups of clones. The first group refers to two code fragments which are similar based on their text. There are three types within the first group as shown in Table 1.

**Type-1 (Exact clones):** Two code fragments are the exact copies of each other except whitespaces, blanks and comments.

**Type-2 (Renamed/Parameterized):** Two code fragments are similar except for names of variables, types, literals and functions.

**Type-3 (Near miss clones/Gapped clones):** Two copied code fragments are similar, but with modifications such as added or removed statements, and the use of different identifiers, literals, types, whitespaces, layouts and comments.

The second group refers to two code fragments which are similar based on their functions. Such clones are also called Type-4 clones as shown in Table 1.

**Type-4 (Semantic clones):** Two code fragments are semantically similar, without being syntactically similar.

### 3. Previous Surveys

Several reviews related to cloned code have been published as shown in Table 2. Most reviews are related to cloned code detection techniques and tools. Burd and Baily evaluate and compare three state-of-the-art clone detection tools (CCFinder, CloneDr, and Covet) and two plagiarism detection tools (JPlag and Moss) in large software applications. This work focuses on benefits of clone identification for preventative maintenance. The paper shows that there is no one single winner. Their work is not comprehensive in coverage of clone detection techniques.

Koschke focuses on clone detection tools and techniques up to 2007. Koschke discusses subareas within the study of clones and summarizing important results and presents open research questions. Koschke compares six clone detectors in terms of recall, precision, clone type, number of candidates, and running time.

Roy and Cordy also survey state-of-the-art in clone detection techniques up to 2007. They provide a taxonomy of techniques, review detection approaches and evaluate them. In addition, this paper discusses how clone detection can assist in other areas of software engineering and how other areas can help clone detection research.

Bellon et al. compare six clone detectors in terms of recall, precision, and space and time requirements. They perform experiments on six detectors using eight large C and Java programs. Their approach to evaluation has become the standards for every a new clone detector introduced since 2007.

Roy et al. classify, compare and evaluate clone detection techniques till 2009. They classify the techniques based on a number of facets. In addition, they evaluate the classified techniques based on a taxonomy of scenarios.

Rysselbergh and Demeyer compare three representative detection techniques: simple line matching, parameterized matching and metric fingerprints. They provide comparison in terms of portability, scalability and the number of false positives. However, they evaluate only on small systems which are smaller than 10 KLOCs (thousands of lines of code) in length.

Ratten et al. perform a systematic review of existing code clone approaches. This review summarizes existing code clone detection techniques, tools and management of clones up to 2012.

Fontana et al. discuss refactored code clone detection and find that certain code quality metrics are improved after the refactoring. They use three clone detection tools and analyze five versions of two open-source systems. The differences in the evaluation results using the three tools are outlined. The impact of clone factoring on different quality metrics is also discussed.

Many methods have been published recently in the literature for detecting code clones. The goal is to discuss, compare and analyze the state-of-the-art tools, and discuss the tools that have not been discussed in previous survey papers. The selected common clone detectors that were covered by previous surveys. Adding clone detectors that are not covered by previous work. Several tools have been excluded from this survey since they are not widely used, are similar to papers that selected based on the texts of the titles, abstracts of the papers, or the use of the same or similar computational techniques. We have found relevant papers to include in this survey by reading other survey papers, by starting from highly cited papers obtained by searching Google Scholar by using keywords such as "software clones", "code clones" and "duplicated code" as shown in Table 4, but then added recent papers that may have lower numbers of citations. We compare and classify techniques and tools considering the types of clones they can detect, the granularity of code fragments they analyze, the transformation they perform on code fragments before comparison, the manner in which they store code fragments to perform comparison, the method used for comparison, the complexity of the comparison process, the outputs they produce, the results they are able to obtain (in terms of precision and recall), and general advantages and disadvantages. We compare this survey with prior surveys in Table 2.

This study particularly differs from previous surveys in our identify the essential strengths and weaknesses of each clone detection technique and tool in contrast to previous studies, which focus on empirically evaluating tools. The goal is not only to compare the current status of the tools and techniques, but also to make an observation indicates that the future potential can be developing a new hybrid technique. The use of evaluation of clone detection techniques based on recall, precision and F-measure metrics, scalability, portability and clone relation in order to choose the right technique for a specific task and several evaluation metrics can be used.
4. CLONE DETECTION PHASES

A clone detector is a tool that reads one or more source files and finds similarities among fragments of code or text in the files. Since a clone detector does not know where the repeated code fragments occur in advance, it must compare all fragments to find them. There are many previous proposed techniques that perform the necessary computation and attempt to reduce the number of comparisons.

We first discuss the phases of clone detection in general. A clone detection technique may focus on one or more of the phases. The first four of phases are shown in Figure 1

4.1 Code Preprocessing

This process removes uninteresting pieces of code, converts source code into units, and determines comparison units. The three major purposes of this phase are given below.

1. Remove uninteresting pieces of code. All elements in the source code that have no bearing in the comparison process are removed or filtered out in this phase.

2. Identify units of source code. The rest of the source code is divided into separate fragments, which are used to check for the existence of direct clone relations to each other. Fragments may be files, classes, functions, begin-end blocks or statements.

3. Identify comparison units. Source units can be divided into smaller units depending upon the comparison algorithm. For example, source units can be divided into tokens.

4.2 Transformation

This phase is used by all approaches except text-based techniques for clone detection. This phase transforms the source code into a corresponding intermediate representation for comparison.

There are various types of representations depending on the technique. The usual steps in transformation are given below.

(1) Extract Tokens. Tokenization is performed during lexical analysis by compiler front ends in programming languages [5][8][9][10]. Each line of source code is converted into a sequence of tokens.

(2) Extract Abstract Syntax Tree. All of the source code is parsed to convert into an abstract syntax tree or parse tree for subtree comparisons [15][44].

(3) Extract PDG. A Program Dependency Graph (PDG) represents control and data dependencies. The nodes of a PDG represent the statements and conditions in a program. Control dependencies represent flow of control information within the program. Data dependencies represent data flow information in a program. A PDG is generated by semantics-aware techniques from the source code for sub-graph comparison [31].

Example 1. Given the source code below, the corresponding PDG is given in Figure 2

```c
int main()
{
    int a = 0;
    int i = 0;
    while (i < 10)
    {
        a = a + i;
    }
}
```
(4) **Other Transformations.** Some techniques apply transformation rules to the source code elements before proceeding with clone detection. These include the CCFinder tool by Kamiya et al. [4], which has transformation rules for C++ remove template parameters. The rule is \( \text{Name} < \text{ParameterList} > \rightarrow \text{Name} \). For example, \( \text{foo}<\text{int}> \) is transformed into \( \text{foo}[4] \).

(5) **Normalization.** This step to remove differences is optional. Some tools perform normalization during transformation. This involves removing comments, whitespaces and differences in spacing as well as normalizing identifiers as shown in Table 3.

### 4.3 Match Detection

The result of transformation or normalization is the input to this phase. Every transformed fragment of code is compared to all other fragments using a comparison algorithm to find similar source code fragments. The output is a set of similar code fragments either in a clone pair list or a set of combined clone pairs in one class or one group as shown in Figure 1. For example, each clone pair may be represented as a quadruplet \((L\text{Begin}, L\text{End}, R\text{Begin}, R\text{End})\), where \(L\text{Begin} \) and \(L\text{End} \) are the left beginning and ending positions of a clone, and \(R\text{Begin} \) and \(R\text{End} \) are the right beginning and ending positions of another clone that following a clone pair [4].

---

**Table 3. Simple Example of Normalization.**

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Normalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>int a, b, c;</td>
<td>int id, id, id;</td>
</tr>
<tr>
<td>a = b + c;</td>
<td>id = id + id;</td>
</tr>
</tbody>
</table>

---

**Fig. 1. Four Phases in CCFinder clone detection tool [4].**

**Fig. 2. Example of Program dependency graph (PDG).**

```c
int main(){ // Method
    int x = 0;
    int y = 2;
    while(x < 10){
        x = x + y;
    }
    while(x < 10){
        x = x + y;
    }
    cout << "Hello World" << endl;
    return 0;
}
```
4.4 Formatting
This step converts a clone pair list obtained by the comparison algorithm in the previous step into a new clone pair list related to the original source code.

4.5 Filtering
Not all clone detectors perform this step. In this phase, code clones are extracted and a human expert filters out the false-positive clones. This step is called Manual Analysis [27]. The false positives can also be filtered out by automated heuristics based on length, diversity or frequency.

4.6 Aggregation
This phase is optional. It can be done in Match Detection phase. To reduce the amount of data, clone pairs can be aggregated into clusters, groups, sets or classes. For example, clone pairs (C1, C2), (C1, C3), and (C2, C3) can be combined into the clone group (C1, C2, C3).

5. CLONE DETECTION TECHNIQUES
Many approaches for detection of code clones have been published in the past years. In this section, the characteristics that can be used to describe clone detection techniques are discussed and follow by a detailed description of a large number of such techniques under appropriate classes.

5.1 Characteristics of Detection Techniques
Several characteristics can be used to describe a clone detection technique. These characteristics depend upon how the technique works. Some characteristics that are used throughout the rest of the paper are given below.

1. Source Transformations or Normalizations. Some transformation or normalization is applied before going to comparison phase in most approaches, but other approaches just remove comments and whitespaces. Some approaches perform detailed transformations or normalizations so that the comparison methods can be applied effectively.

2. Source Code Representation. A comparison algorithm represents code using its own format or representation scheme. Most comparison algorithms use appropriate code that results from the transformation or normalization phase.

3. Comparison Granularity. Different comparison algorithms look at code at different levels of granularity such as lines, tokens, nodes of abstract syntax trees (ASTs) or nodes of a program dependency graph (PDG).

4. Comparison Algorithms. There are many algorithms that are used to detect clones such as suffix-tree algorithms [5] [10], sequence matching algorithms [4], dynamic pattern matching algorithms [12] and hash-value comparison algorithms [13].

5. Computational Complexity. A technique must be expandable to detect clones in millions of lines of source code in a large system. In other words, it must be computationally efficient. The complexity of a tool is based on the kind of transformations or normalizations performed in addition to the comparison algorithm.

6. Clone Types. Some techniques detect Type-1 clones while others find Type-1 or Type-2 or Type-3 clones or may even detect all types of clones.

(7) Language Independence. A language-independent tool can work on any system without any concern. Thus, we should be aware of any language-dependent issues for our chosen method.

(8) Output of Clones. Some techniques report clones as clone pairs while others return clones as clone classes or both. Clone classes are better than clone pairs for software maintenance. Clone classes, which are reported without filtering are better than the ones that are returned after filtering because the use of clone classes reduces the number of comparisons and amount of clone data that needs to be reported.

5.2 Categories of Detection Techniques
Detection techniques are categorized into four classes. The textual, lexical, syntactic and semantic classes are discussed. Syntactic approaches can be divided into tree-based and metric-based techniques and semantic approaches can be divided into PDG-based and hybrid techniques as shown in Table 4. In this section, state-of-the-art in clone detection techniques are described and compared, under these classes and subclasses.

5.2.1 Textual Approaches. Text-based techniques compare two code fragments and declare them to be clones if the two code fragments are literally identical in terms of textual content. Text-based clone detection techniques generate fewer false positives, are easy to implement and are independent of language. Text-based clone detection techniques perform almost no transformation to the lines of source code before comparison. These techniques detect clones based on similarity in code strings and can find only Type-1 clones.

In this section, several well-known textual approaches or text-based techniques as shown in Table 5 are discussed. These include Dup [8] by Baker, Duploc tool [15], Ducasse et al. [26], Koschke et al. [14], NICAD by Roy and James [11] and SSD by Seunghak and Jeong [12].

5.2.1.1 Dup by Baker. Dup [8] reads source code line by line in the lexical analysis phase. Dup uses normalization, which removes comments and whitespaces and also handles identifier renaming. It hashes each line for comparison among them and extracts matches by a suffix-tree algorithm. The purpose of this tool is to find maximal sections of code that are either exact copies or near miss clones of each other. The Dup tool can also be classified as a token-based technique since it tokenizes each line for line-by-line matching.

To explain Dup’s technique, it is necessary to introduce the term parameterized string (p-string), which is a string over the union of two alphabets, say Σ and Π. It also introduces the notion of parameterized match (p-match), which refers to the process in which a p-string is transformed into another p-string by applying a renaming function from the symbols of the first p-string to the symbols of the second p-string. Parameterized matches can be detected using parameterized strings or p-strings, which are strings that contain ordinary characters from an alphabet Σ, and parameter characters from a finite alphabet Π. Dup implements a p-match algorithm. The lexical analyzer produces a string containing non-parameter symbol and zero or more parameter symbols. When sections of code match except for the renaming of parameters, such as variables and constants, p-match occurs. Exact match can be detected using a plain suffix tree.

The Dup algorithm encodes a p-string in the following way. The first appearance of each parameter symbol is replaced by
Table 4. Techniques for Clone Detection.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Technique</th>
<th>Tool/Author</th>
<th>Year</th>
<th>Reference</th>
<th>Citation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>NICAD</td>
<td>2008</td>
<td>[1]</td>
<td>183</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SDD</td>
<td>2005</td>
<td>[12]</td>
<td>26</td>
</tr>
<tr>
<td>Lexical</td>
<td>Token</td>
<td>CCFinder</td>
<td>2002</td>
<td>[14]</td>
<td>1126</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Boreas</td>
<td>2012</td>
<td>[9]</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FRISC</td>
<td>2012</td>
<td>[65]</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CDSW</td>
<td>2013</td>
<td>[66]</td>
<td>9</td>
</tr>
<tr>
<td>Syntactic</td>
<td>Tree</td>
<td>CloneDr</td>
<td>1998</td>
<td>[13]</td>
<td>1003</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Wahler et al.</td>
<td>2004</td>
<td>[34]</td>
<td>118</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Koschke et al.</td>
<td>2006</td>
<td>[1]</td>
<td>189</td>
</tr>
<tr>
<td>Metric</td>
<td>Token</td>
<td>Mayvand et al.</td>
<td>1996</td>
<td>[29]</td>
<td>469</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Kontogiannis et al.</td>
<td>1996</td>
<td>[37]</td>
<td>254</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Kodhai et al.</td>
<td>2010</td>
<td>[19]</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Kanika et al.</td>
<td>2013</td>
<td>[29]</td>
<td>3</td>
</tr>
<tr>
<td>Semantic</td>
<td>Graph</td>
<td>Duplix</td>
<td>2001</td>
<td>[31]</td>
<td>474</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GPLAG</td>
<td>2006</td>
<td>[32]</td>
<td>239</td>
</tr>
<tr>
<td>Hybrid</td>
<td></td>
<td>Higo and Kasumoto</td>
<td>2009</td>
<td>[39]</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ConQAT</td>
<td>2011</td>
<td>[39]</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Agrawal et al.</td>
<td>2013</td>
<td>[10]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Funaro et al.</td>
<td>2010</td>
<td>[10]</td>
<td>12</td>
</tr>
</tbody>
</table>

zero and each subsequent appearance of a parameter symbol is substituted by the distance from the previous appearance of the same symbol. The non-parameter symbol is not changed as shown in Example 2 below. The $Dup$ algorithm uses Definition 4 and Proposition 1, given below from [31, 32], to represent parameter strings in a $p$-suffix tree. In Definition 4, the $f$ function, called $transform$, computes the $j$-th symbol value of $p$-suffix($S,i$) in constant time from $j$ and $(j+i-1)$.

A parameterized suffix tree ($P$-suffix tree) is a data structure for generalization of suffix trees for strings. $P$-suffix encoding requires that a $p$-string $P$ and a $p$-string $P'$ are p-match of each other if and only if $prev(P) = prev(P')$, where $prev$ is the resulting encoding of $P$. For example, when we have a $p$-string $T$ that has the same encoding as the $p$-string $P$, and $T$ and $P$ are a $p$-match. Therefore, $prev$ is used to test for $p$-matches. If $P$ is a $p$-string pattern and $P'$ is a $p$-string text, $P$ has a $p$-match starting at position $i$ of $T$ if and only if $prev(P)$ is a prefix of $p$-suffix($T,i$).

Definition 4. If $b$ belongs to alphabet $\Sigma$ union alphabet $\Pi$, $f(b,j)=b$ if $b$ is a nonnegative integer larger than $j-1$, and otherwise, $f(b,j)=b$ [60].

Proposition 1. Two $p$-strings $P$ and $T$ $p$-match when $prev(P) = prev(T)$. Also, $P < T$ when $prev(P) < prev(T)$ and $P > T$ when $prev(P) > prev(T)$ [60].

Example 2. Let $P = yxbxxbyS$, where $x$ and $y$ are parameter symbols and $b$ and $S$ are non-parameter symbols. The $p$-suffixes are $0b021b6y$, $0b21b0S$, $b01b0S$, $0ib0S$, $0b0S$, $b0S$, $0S$, and $S$ as shown in Figure 3. The $Dup$ algorithm finds parameterized duplication by constructing a $p$-suffix tree and recursing over it.

In the example in Figure 3, $0$ is detected as duplicated in $0b0S$, $0ib0S$, $0b21b0S$, and $0b021b6y$. $b0$ is detected as duplicated also in $b01b0S$.

5.2.1.2 Duploc by Ducasse et al. $Duploc$ [15] is also a text-based clone detection technique. $Duploc$ uses an algorithm that has two steps. The first step transforms source files into normalized files after eliminating noise including all whitespaces, comments.
and converts all characters to lower case. Noise elimination reduces false positives by removing common constructs. It also reduces false negatives by removing insignificant differences between code clones. The second step compares normalized files line-by-line using a simple string-matching algorithm. The hits and misses that the comparison produces are stored in a matrix and are visualized as a dotplot [17][29][30]. The computational complexity is $O(n^2)$ for an input of $n$ lines. Preprocessing transformed lines reduces the search space. Each line is hashed into one of a number of buckets. Every occurrence of the same hashed line value is placed in the same hash bucket. Duploc is able to detect a significant amount of identical code duplicates, but it is not able to identify renaming, deletion and insertion. Duploc does not perform lexical analysis or parsing. The advantage of the character-based technique that Duploc uses is its high adaptability to diverse programming languages.

Ducasse et al. [26] add one more step to Duploc, a third step of filters. This step extracts interesting patterns in duplicated code such as gap size, which is the length of a non-repeated subsequence between a clone pair. For example, if the line sequences 'abcghdjg' and 'abcfkhjg' are compared, the gap is of length 4 between the lengths of the two non-duplicated subsequences gdhj and jhfn are 4. False positives are averted by removing noise and by filtering. The filter step uses two criteria [26]. 1) Minimum length: It is the smallest length of a sequence to be important. 2) Maximum gap size: It is the largest gap size for sequences to be obtained by copy-pasting from one another. The algorithm implements filtering in a linear amount of single matches. Ducasse's tool uses lexical analysis to remove comments and whitespaces in code and finds clones using a dynamic pattern matching (DPM) algorithm. The tool's output is the number lines of code clone pairs. It partitions lines using a hash function for strings for faster performance. The computational complexity is $O(n^2)$, where $n$ is the input size.

Koschke et al. [14] prove that Duploc detects only copy-pasted fragments that are exactly identical. It cannot detect Type-3 clone or deal with modifications and insertions in copy-pasted code, called tolerance to modifications. Duploc cannot detect copy-pasted bugs [14] because detecting bugs requires semantic information and Duploc detects just syntactic clones.

5.2.1.3 NICAD by Roy and James. Roy and James [11] develop a text-based code clone detection technique called Accurate Detection of Near-miss Intentional Clones (NICAD). The NICAD tool [13][18] uses two clone detection techniques: text-based and abstract syntax tree-based, to detect Type-1, Type-2 and Type-3 cloned code. The structures of the two approaches complement each other, overcoming the limitations of each technique alone. NICAD has three phases. 1) A parser extracts functions and performs pretty-printing that breaks different fragments of a statement into lines. 2) The second phase normalizes fragments of a statement to ignore editing differences using transformation rules. 3) The third phase checks potential clones for renaming, filtering or abstraction using dynamic clustering for simple text comparison of potential clones. The longest common subsequence (LCS) algorithm is used to compare two potential clones at a time. Therefore, each potential clone must be compared with all of the others, which makes the comparison expensive.

NICAD detects near-misses by using flexible pretty-printing. Using agile parsing [50] and the Turing eXtender Language (TXL) transformation rules [51] during parsing and pretty-printing, it can easily normalize code. By adding normalization to pretty-printing, it can detect near-miss clones with 100% similarity. After the potential clones are extracted, the LCS algorithm compares them. The NICAD tool uses percentage of unique strings (PUS) for evaluation. Equation (1) computes the percentage of unique strings for each possible clone.

If $PUS = 0\%$, the potential clones are exact clones; otherwise, if $PUS$ is more than $0\%$ and below a certain threshold, the potential clones are near-miss clones.

$$PUS = \frac{\text{Number of Unique Strings} \times 100}{\text{Total Number of Strings}}$$

NICAD finds exact matches only when the PUS threshold is $0\%$. If the PUS threshold is greater than $0\%$, clone 1 is matched to clone 2 if and only if the size, in terms of number of lines, of the second potential clone is between $\text{size (clone 1)} - \text{size (clone 2)} \times \text{PUS / 100}$ and $\text{size (clone 1)} + \text{size (clone 2)} \times \text{PUS / 100}$.

NICAD can detect exact and near-miss clones at the block level of granularity. NICAD has high precision and recall [15]. It can detect even some exact function clones that are not detected by the exact matching function used by a tree-based technique [13][18]. NICAD exploits the benefits of a tree-based technique by using simple text lines instead of subtree comparison to obtain good space complexity and time.

5.2.1.4 SDD by Seunghak and Jeong. Seunghak and Jeong [12] use a text-based code clone detection technique implemented in a tool called the Similar Data Detection (SDD), that can be used as an Eclipse plug-in. Eclipse is an integrated development environment (IDE) [49]. The Eclipse IDE allows the developer to extend the IDE functionality via plug-ins. SDD detects repeated code in large software systems with high performance. It also detects exact and similar code clones by using an inverted index [57] and an index data structure using a $n$ neighbor distance algorithm [58]. The mean nearest neighbor distance is:

$$\text{Nearest Neighbor Distance} = \frac{\sum_{i = 1}^{N} \text{Min}(d_{ij})}{N}$$

where $N$ is the number of points and $\text{Min}(d_{ij})$ is the distance between each point and its nearest neighbor. SDD is very powerful for detection of similar fragments of code in large systems because use of inverted index decreases SDD complexity.

5.2.2 Summary of Textual Approaches. In this section, several textual approaches for clone detection are discussed. Dup [8] uses a suffix-tree algorithm to find all similar subsequences using hash values of lines, characters or tokens. The complexity of computation is $O(n)$ where $n$ is the input length of the sequence. Duploc [15] uses a dynamic pattern matching algorithm to find a longest common subsequence between two sequences. NICAD [11] uses the Longest Common Subsequence algorithm to compare two lines of potential clones and produces the longest sequence. The LCS algorithm compares only two sequences at a time. Therefore, the number of comparisons is high because each sequence must be compared with all of the other sequences. SDD [12] uses the $n$-neighbor distance algorithm to find near-miss clones. It may lead to detection of false positives.
Text-based techniques have limitations as follows \[10, 8, 17\].

1. Identifier renaming cannot be handled in a line-by-line method.
2. Code fragments with line breaks are not detected as clones.
3. Adding or removing brackets can create a problem when comparing two fragments of the code where one fragment has brackets and the second fragment does not have brackets. 
4. The source code cannot be transformed in text-based approaches. Some normalization can be used to improve recall without sacrificing high precision \[4\].

5.2.3 Lexical Approaches. Lexical approaches are also called token-based clone detection techniques. In such techniques, all source code lines are divided into a sequence of tokens during the lexical analysis phase of a compiler. All tokens of source code are converted back into token sequence lines. Then the token sequence lines are matched. In this section, several state-of-the-art token-based techniques as discussed in Table 6 are discussed. These include CCFinder \[3\] by Kamiya et al., CP-Miner \[9,15\] by Zhenmin et al., Boreas \[9\] by Yong and Yao, FRISC \[9\] by Murakami et al., and CD SW \[9\] by Murakami et al. These techniques as examples are chosen because they are among the best such techniques and can detect various types of clones with higher recall and precision than a text-based technique.

5.2.3.1 CCFinder by Kamiya et al. Kamiya et al. \[3\] develop a suffix tree-matching algorithm called CCFinder. CCFinder has four phases. 1) A lexical analyzer removes all whitespaces between tokens from the token sequence. 2) Next, the token sequence is sent to the transformation phase that uses transformation rules. It also performs parameter replacement where each identifier is replaced with a special token. 3) The Match Detection phase detects equivalent pairs as clones and also identifies classes of clones using suffix tree matching. 4) The Formatting phase converts the locations of clone pairs to line numbers in the original source files.

CCFinder applies several metrics to detect interesting clones. These metrics are given below. 1) The length of a code fragment that can be used by the number of tokens or the number of lines of the code fragments. 2) Population size of a clone class: It is the number of elements in a clone class. 3) Combination of the number of tokens and the number of elements in a clone class for estimating which code portion could be refactored. 4) Coverage of code clone: It is either the percentage of lines or files that include any clones. It also optimizes the sizes of programs to reduce the complexity of the token matching algorithm. It produces high recall whereas its precision is lower than that of some other techniques \[10\]. CCFinder accepts source files written in one programming language at a time.

The line-by-line method used by the Duploc tool \[15\] discussed earlier, cannot recognize or detect clones with line break relocation, the layout of the code is changed. CCFinder performs a more suitable transformation than the line-by-line method \[3\]. CCFinder can also handle name changes, which the line-by-line approach cannot handle. However, CCFinder or a token-based technique takes more CPU time and more memory than line-by-line comparison \[8,35,17,5\]. CCFinder uses a suffix tree algorithm, and so it cannot handle statement insertions and deletions in code clones \[7\].

5.2.3.2 CP-Miner by Li et al. Li et al. \[8,35\] use a token-based technique to detect code clones and clones related to bugs in large software systems. Their system, CP-Miner, searches for copy-pasted code blocks using frequent subsequence mining \[24\]. CP-Miner implements two functions.

(i) Detecting copy-pasted code fragments. CP-Miner converts the problem into a frequent subsequence mining problem by parsing source code to build a database containing a collection of sequences. It then implements an enhanced version of the CloSpan algorithm \[24\] which is used to help satisfy gap constraints in frequent subsequences. Each similar statement is mapped to the same token. Similarly, each function name is mapped onto the same token. All tokens of a statement are hashed using the hashpjw hash function \[25\]. After parsing the source code, CP-Miner produces a database with each sequence representing a fragment of code. A frequent sub-sequence algorithm and CloSpan are applied to this database to find frequent sub-sequences. CP-Miner composes larger copy-pasted segments both of real copy-paste and false positives in groups. It then checks neighboring code segments of each duplicate to see if they are a copy-pasted group as well. If so, the two groups are merged. The larger group is checked again against the false positives.

(ii) Finding copy-paste related bugs. Frequently, programmers forget to rename identifiers after copy-pasting. Unchanged identifiers are detected by a compiler and reported as “errors”. These errors become unobserved bugs that can be very hard to detect by a detector. Therefore, CP-Miner uses an UnchangedIDRatio threshold to detect bugs.

\[
\text{UnRenamedIDRate} = \frac{\text{NumUnchangedID}}{\text{TotalUnchangedID}} \tag{3}
\]

where UnRenamedIDRate is the percentage of unchanged identifiers, NumUnchangedID is the number of unchanged identifiers and TotalUnchangedID is the total number of identifiers in a given copy-pasted fragment. The value of UnRenamedIDRate can be any value in the range 0 and 1. If UnRenamedID Rate is 0, it means all occurrences of identifiers have been changed, and if UnRenamedIDRate is 1, it means all occurrences of the identifier remain unchanged.

CP-Miner can only detect forgot-to-change bugs. This means if the programmer has forgotten to modify or insert some extraneous statements to the new copy-pasted segment, CP-Miner would not detect the bug because the changed code fragments are now too different from the others \[8,35\]. This approach can detect similar sequences of tokenized statements and avert redundant comparisons, and as a result, it detects code clones efficiently, even in millions of code lines.

CP-Miner overcomes some limitations of CCFinder and detects more copy-pasted segments than CCFinder does. However, CCFinder does not detect code clones that are related to bugs as CP-Miner does because CP-Miner uses an unchanged ratio threshold. CCFinder does not completely filter false positives and it detects many tiny cloned code blocks which seem to be predominantly false positives. Because CP-Miner handles statement insertions and modifications, CP-Miner can detect 17-52% more code clones than CCFinder. Unfortunately, the frequent subsequence mining algorithm that CCFinder uses has two limitations because it divides a long sequence into sub-sequences. First, some frequent subsequences of two or more statement blocks may be lost. Second, it is hard to choose the size of short sequences because if the size is too short, the information may be lost; if the size is too long, the mining algorithm may be very slow \[8,35\].
Table 5. Summary of Textual Approaches.

<table>
<thead>
<tr>
<th>Author/Tool</th>
<th>Transformation</th>
<th>Code Representation</th>
<th>Comparison Method</th>
<th>Complexity</th>
<th>Granularity</th>
<th>Types of Clone</th>
<th>Language Independence</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dup [12]</td>
<td>Remove whitespace and comments</td>
<td>Parameterized string matches</td>
<td>Suffix-tree based token matching</td>
<td>(O(n + m)) where (n) is number of input lines and (m) is number of matches</td>
<td>Token of lines</td>
<td>Type-1, Type-2</td>
<td>Needs lexer</td>
<td>Text</td>
</tr>
<tr>
<td>Duploc [13]</td>
<td>Removes comments and all white space</td>
<td>Sequence of lines</td>
<td>Dynamic Pattern Matching</td>
<td>(O(n^*)) where (n) is number of input lines</td>
<td>Line</td>
<td>Type-1, Type-2</td>
<td>Needs lexer</td>
<td>Text</td>
</tr>
<tr>
<td>NICAD [14]</td>
<td>Pretty-printing</td>
<td>Methods (Segment sequences)</td>
<td>Longest Common Subsequence (LCS)</td>
<td>(O(n^2)) worst case time and space</td>
<td>Text</td>
<td>Type-1, Type-2, Type-3</td>
<td>Needs parser</td>
<td>Text</td>
</tr>
<tr>
<td>SMR [15]</td>
<td>No transformation</td>
<td>Inverted index and index</td>
<td>N-neighbor distance</td>
<td>(O(n))</td>
<td>Chunks of source code</td>
<td>Type-1, Type-2, Type-3</td>
<td>No lexer/parser needs</td>
<td>Visualization similar of code</td>
</tr>
</tbody>
</table>


The computation of the Count Environment (CE) is divided into three stages. The first stage is Naïve Counting, which counts the number variables used and defined in the environments. The second stage is In-statement counting, which counts the number of regular variables as well as the variables used as if-predicates and array subsripts, and variables that are defined by expressions with constants. The third stage is Inter-statement Counting, which counts variables used inside a first-level loop, second level loop or third level loop. 2) Count Vector (CV), which is produced using \(m\) (m-dimensional Count Vector) CE’s. The i-th dimension in the CV is the number of the variable in the i-th CE. CV is also called a characteristic vector. 3) Counting Matrix (CM), which contains all \(n\) (n-variables) CVs in code fragments and is an \(n \times m\) Count Matrix. Boreas uses the cosine of the two vectors angle to compare similarity:

\[
Sim(v_1, v_2) = \cos(\alpha) = \frac{\sum_{i=1}^{n} v_1^i \times v_2^i}{\sqrt{\sum_{i=1}^{n} v_1^i \times v_1^i} \times \sqrt{\sum_{i=1}^{n} v_2^i \times v_2^i}}
\]

(4)

where \(Sim\) is the cosine similarity between two vectors \(v_1\) and \(v_2\) and \(\alpha\) is the angle between them. The similarity between two fragments is measured by an improved proportional similarity function. This function compares the CVs of keywords and punctuations marks. \(PropSimilarity\) is proportional similarity between \(C_1\) and \(C_2\), which are two occurrences counts:

\[
PropSimilarity = \frac{1}{(C_1 + 1)} + \frac{C_2}{(C_1 + 1)}.
\]

(5)

The function prevents incorrect zero similarity. Boreas is not able to detect code clones of Type-3. Agrawal et al. [15] extend Boreas to detect clones by using a token-based approach to match clones with one variable or a keyword and easily detect Type-1 and Type-2 clones; they use a textual approach to detect Type-3 clones. Since Agrawal et al.’s approach combines two approaches, it is a hybrid approach.

5.2.3.4 FRISC by Murakami et al. Murakami et al. [65] develop a token-based technique called FRISC which transforms every repeated instruction into a special form and uses a suffix array algorithm to detect clones. FRISC has five steps. 1) Performing lexical analysis and normalization, which transforms source files into token sequences and replaces every identifier by a special token. 2) Generating statement hash, which generates a hash value for every statement between “,” “[”, and “]” with every token included in a statement. 3) Folding repeated instructions, which identifies every repeated subsequence and divides into the first repeated element and its subsequent repeated elements. Then, the repeated subsequences are removed and their numbers of tokens are added to their first repeated subsequence of elements. 4) Detecting identical hash subsequences, which detects identical subsequences from the folded hash sequences. If the sum of the numbers of tokens is smaller than the minimum token length, they are not considered clones. 5) Mapping identical subsequences to the original source code, which converts clone pairs to original source code.

FRISC supports Java and C. The authors performed experiments with eight target software systems, and found that the precision with folded repeated instructions is higher than the precision without by 29.8%, but the recall decreases by 2.9%.

Fig. 4. CP-Miner Process Steps.
FRISC has higher recall and precision than CCFinder but the precision is lower than CloneDr \cite{13} and CLAN \cite{21}.

5.2.3.5 CDSW by Murakami et al. Murakami et al. \cite{66} develop another token-based technique, which detects Type-3 clones (gapped clones) using the Smith-Waterman algorithm \cite{68}, called CDSW. It eliminates the limitations of AST-based and PDG-based techniques which, require much time to transform source code into ASTs or PDGs and compare among them. CDSW has five steps. 1) Performing lexical analysis and normalization, which is the same as the first step of FRISC. 2) Calculating hash values for every statement, which is the same as FRISC. 3) Identifying similar hash sequences, which identifies similar hash sequences using the Smith-Waterman Algorithm. 4) Identifying gapped tokens using Longest Common Subsequences (LCS) to identify every sequence gap. 5) Mapping identical subsequences to the source code, which converts clone pairs to the original source code. It is also performs the same fifth step as FRISC.

Since Bellon’s references, which are built by manually confirming a set of candidates to be clones or clone pairs that are judged as correct \cite{64}, do not contain gapped fragments, Murakami et al. enhance the clone references by adding information about gapped lines. Murakami et al. calculate recall, precision, and f-measure using Bellon’s \cite{64} and their own clone references resulting in improved recall, precision, and f-measure. recall increased by 4.1%, precision increased by 3.7%, and f-measure increased by 3.8% in the best case. recall increased by 0.49%, precision increased by 0.42% and f-measure increased by 0.43% in the worst case \cite{66}. The results are different because CDSW replaces all variables and identifiers with special tokens that ignore their types. Because CDSW does not normalize all variables and identifiers, it cannot detect clones that have different variable names \cite{66}.

5.2.4 Summary of Lexical Approaches. The suffix-tree based token matching algorithm used by CCFinder finds all similar subsequences in a transformed token sequence. CCFinder cannot detect statement insertions and deletions in copypasted code. It does not completely eliminate false positives. The frequent subsequence mining technique used by CP-Miner discovers frequent subsequences in a sequence database. A frequent subsequence mining technique avoids unnecessary comparisons, which makes CP-Miner efficient. CP-Miner detects 17%-52% more code clones than CCFinder. A limitation of a frequent subsequence mining algorithm is that a sequence database is needed. Boreas works fast by using two functions cosine similarity and proportional similarity. FRISC detects more false positives than the other tools but misses some clone references \cite{65}. CDSW’s accuracy is based on the match, mismatch and gap parameters. If these parameters are changed, the results are different.

Token-based techniques have limitations as follows. 1) Token-based techniques depend on the order of program lines. If the statement order is modified in duplicated code, the duplicated code will not be detected. 2) These techniques cannot detect code clones with swapped lines or even added or removed tokens because the clone detection is focused on tokens. 3) Token-based techniques are more expensive in time and space complexity than text-based techniques because a source line contains several tokens.

5.2.5 Syntactical Approaches. Syntactical approaches are categorized into two kinds of techniques. The two categories are tree-based techniques and metric-based techniques. A list of syntactical techniques found in the literature is shown in Table 7. In this section, several common tree-based and metric-based techniques are discussed. For the purpose of this study, we choose CloneDR \cite{13} by Baxter et al., Wahler \cite{54}, Koschke \cite{14}, Jiang \cite{28}, Mayrand et al. \cite{20}, Kontogiannis et al. \cite{37}, Kodai, et al. \cite{19}, Abdul-El-Hafiz \cite{48} and Kanika et al. \cite{29}.

5.2.5.1 Tree-based Clone Detection Techniques. In these techniques, the source code is parsed into an abstract syntax tree (AST) using a parser and the sub-trees are compared to find cloned code using tree-matching algorithms. CloneDr by Baxter et al Baxter et al. \cite{13} use a tree-based code clone detection technique implemented in a tool called CloneDr. It can detect exact clones, near miss clones and refactored code using an AST. After the source code is parsed into an AST, it finds clones by applying three main algorithms. The first algorithm detects sub-tree clones, the second algorithm detects variable-size sequences of sub-tree clones such as sequences of declarations or statements, and the third algorithm finds more complex near-miss clones by generalizing other clone combinations. The method splits sub-trees using a hash function and then compares sub-trees in the same bucket. The first algorithm finds sub-tree clones and compares each sub-tree with other sub-trees. Near-miss clones that cannot be detected by comparing sub-trees can be found using similarity computation:

\[ \text{Similarity} = \frac{2SN}{(2SN + \text{LEFT} + \text{RIGHT})} \]  

where \( SN \) is the number of shared nodes, \( \text{LEFT} \) is the number of nodes in sub-tree1 and \( \text{RIGHT} \) is the number of nodes in sub-tree2. The second algorithm finds clone sequences in ASTs. It compares each pair of sub-trees and looks for maximum length sequences. The third algorithm finds complex near-miss clones. It abstracts each pair of clones after all clones are detected.

CloneDr cannot detect semantic clones. Text-based techniques do not deal with modifications such as renaming of identifiers since there is no lexical information. Tree-based techniques may produce false positives since two fragments of the sub-tree may not be duplicated. Because a tree-based method hashes subtrees, it cannot detect duplicated code which has modifications. Wahler et al Wahler et al. \cite{54} detect clones which are represented as an abstract syntax tree (AST) in XML by applying frequent itemset mining. Frequent itemset mining is a data mining technique that looks for sequences of actions or events that occur frequently. An instance is called a transaction, each of which has a number of features called items. This tool uses frequent itemsets to identify features in large amounts of data using the Apriori algorithm \cite{52}. For each itemset, they compute its support count, which is the frequency of occurrence of an itemset or the number of transactions in which it appears:

\[ \sigma(I) = \frac{|\{T \in D | I \subseteq T\}|}{|D|} \geq \sigma \]  

where \( T \) is a transaction, \( I \) is an itemset, which is a subset of the transaction \( T \), and \( D \) is a database. If an itemset’s frequency is more than a certain given support count \( \sigma \), it is called a frequent itemset.

There are two steps to find frequent itemsets. The first step is the join step. The first step finds \( L_k \), which are frequent itemsets
of size $k$. A set of candidate $k$-itemsets is generated by combining $L_{k-1}$ with itself. The second step is the prune step, which finds frequent $k$-itemsets from $C_k$. This process is repeated until no more frequent $k$-itemsets are found. In this approach, the statements of a program become items in the database $D$. Clones are a sequence of source code statements that occur more than once. Therefore, the support count is $\sigma = \frac{1}{n}$. Let there be statements $b_1...b_k$ in a program. The join step combines two frequent $(k-1)$-itemsets of the form $I_1 = b_1...b_{k-1}, I_2 = b_2...b_{k-1}$. Koschke et al Koschke et al. [14] also detect code clones using an abstract syntax tree (AST). Their method finds syntactic clones by pre-order traversal, applies suffix tree detection to find full subtree copies, and decomposes the resulting Type-1 and Type-2 token sequences. This approach does not allow structural parameters. It can find Type-1 and Type-2 clones in linear time and space. A set of candidate $k$-itemsets is generated by combining $C_{k-1}$ with itself. The second step is the prune step, which finds frequent $k$-itemsets from $C_k$. This process is repeated until no more frequent $k$-itemsets are found. In this approach, the statements of a program become items in the database $D$. Clones are a sequence of source code statements that occur more than once. Therefore, the support count is $\sigma = \frac{1}{n}$. Let there be statements $b_1...b_k$ in a program. The join step combines two frequent $(k-1)$-itemsets of the form $I_1 = b_1...b_{k-1}, I_2 = b_2...b_{k-1}$. Koschke et al Koschke et al. [14] also detect code clones using an abstract syntax tree (AST). Their method finds syntactic clones by pre-order traversal, applies suffix tree detection to find full subtree copies, and decomposes the resulting Type-1 and Type-2 token sequences. This approach does not allow structural parameters. It can find Type-1 and Type-2 clones in linear time and space. AST-based detection can be used to find syntactic clones with more effort than Token-based suffix trees and with low precision. AST-based detection also scales worse than Token-based detection. Token-based suffix tree clone detectors can be adapted to a new language in a short time whereas using AST needs a full abstract syntax tree and sub-tree comparison method. Using abstract syntax suffix trees [13] detects clones in less time.

**Table 6. Summary of Lexical Approaches.**

<table>
<thead>
<tr>
<th>Author/Tool</th>
<th>Transformation</th>
<th>Code Representation</th>
<th>Comparison Method</th>
<th>Complexity</th>
<th>Granularity</th>
<th>Types of Clone</th>
<th>Language Independence</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCFinder [21]</td>
<td>Remove whitespace, comments, and perform parameter replacement</td>
<td>Normalized sequences and parameterized tokens</td>
<td>Suffix-tree based on token matching</td>
<td>$O(n)$ where $n$ is size of source file</td>
<td>Token</td>
<td>Type-1, Type-2</td>
<td>Needs lexer and transformation rules</td>
<td>Clone pairs/Clone classes</td>
</tr>
<tr>
<td>CP-Mines [19]</td>
<td>Map each statement/identifier to a number with similar statements/identifiers</td>
<td>Basic blocks</td>
<td>Frequent subsequence mining technique</td>
<td>$O(n^2)$ where $n$ is number of code lines</td>
<td>Sequence of tokens</td>
<td>Type-1, Type-2</td>
<td>Needs parser</td>
<td>Clone pairs</td>
</tr>
<tr>
<td>Boreas [22]</td>
<td>Variables matching based on other characteristics</td>
<td>Suffix array</td>
<td>Cosine Similarity Function</td>
<td>N/A</td>
<td>Token sequences</td>
<td>Type-1, Type-2, Type-3</td>
<td>Needs lexer</td>
<td>Clustering</td>
</tr>
<tr>
<td>FRISC [65]</td>
<td>Remove whitespaces, comments, mapping from transformed sequence into original, and parameter replace</td>
<td>Hash sequence of tokens</td>
<td>Suffix array</td>
<td>N/A</td>
<td>Token sequences</td>
<td>Type-1, Type-2, Type-3</td>
<td>Needs lexer</td>
<td>Clone pairs/Clone classes</td>
</tr>
<tr>
<td>CDSW [66]</td>
<td>Remove whitespaces, comments, mapping from transformed sequence into original, and parameter replace</td>
<td>Hash values for every statement</td>
<td>Smith-Waterman Alignment</td>
<td>$O((nm)^2)$ where $n$ is length of first token sequences and $m$ is length of second token sequences</td>
<td>Token sequences</td>
<td>Type-1, Type-2, Type-3</td>
<td>Needs lexer</td>
<td>Clone pairs</td>
</tr>
</tbody>
</table>

Hotta et al Hotta et al. [67] compare and evaluate methods for detection of coarse-grained and fine-grained unit-level clones. They use a coarse-grained detector that detects block-level clones from given source files. Their approach has four steps. 1) Lexical and syntactic analysis to detect all blocks from the given source files such as classes, methods and block statements. 2) Normalization of every block detected in the previous step. This step detects Type-1 and Type-2 clones. 3) Hashing every block using the hashCode() function java.lang.String. 4) Grouping blocks based on their hash values. If two normalized blocks have the same hash value, they are considered equal to each other as in Figure 5. The detection approach has high accuracy, but Hotta et al.’s method, which is coarse-grained does not have high recall compared to fine-grained detectors, does not tackle gapped code clones, and detects fewer clones. Their approach is much faster than a fine-grained approach, since the authors use hash values of texts of blocks. However, using a coarse-grained approach alone is not enough because it does not have more detailed information about the clones. A fine-grained approach must be used as a second stage after a coarse-grained approach.

**5.2.5.2 Metric-based clone detection techniques.**

In metric-based clone detection techniques, a number of metrics are computed for each fragment of code to find similar fragments by comparing metric vectors instead of comparing code or ASTs directly. Seven software metrics have been used by different authors [23, 24, 25, 26].

1. Number of declaration statements,
2. Number of loop statements,
3. Number of executable statements,
4. Number of conditional statements,
5. Number of return statements,
6. Number of function calls, and
7. Number of parameters.

All of these metrics are computed and their values are stored in a database [19]. Pairs of similar methods are also detected by comparison of the metric values, which are stored in the same database.
Mayrand et al. [20] compute metrics from names, layouts, expressions and control flows of functions. If two functions’ metrics are similar, the two functions are considered to be clones. Their work identifies similar functions but not similar fragments of code. In reality, similar fragments of codes occur more frequently than similar functions.

First, source code is parsed to an abstract syntax tree (AST). Next, the AST is translated into an Intermediate Representation Language (IRL) to detect each function. This tool reports as clone pair two function blocks with similar metrics values. Patenaude et al. [35] extend Mayrand’s tool to find Java clones using a similar metric-based algorithm. Kontogiannis et al. [36] propose a way to measure similarity between two pieces of source code using an abstract patterns matching tool. Markov models are used to calculate dissimilarity between an abstract description and a code fragment. Later, they propose two additional methods for detecting clones [37]. The first method performs numerical comparison of the metric values that categorize a code fragment to begin-end blocks. The second approach uses dynamic programming to compute and report begin-end blocks using minimum edit distance. This approach only reports similarity measures and the user must go through block pairs and decide whether or not they are actual clones. Kodhai et al. [19] combine a metric-based approach with a text-based approach to detect functional clones in C source code. The process of clone detection has five phases. 1) The Input and Pre-processing step parses files to remove pre-processor statements, comments, and whitespaces. The source code is rebuilt to a standard form for easy detection of similarity of the cloned fragments. 2) Template conversion is used in the textual comparison of potential clones. It renames data types, variables, and function names. 3) Method identification identifies each method and extracts them. 4) Metric Computation. 5) Type-1 and Type-2 clone detection. The text-based approach finds clones with high accuracy and reliability, but the metric-based approach can reduce the high complexity of the text-based approach by using computed metrics values. The limitation of this method is that it just detects Type-1 and Type-2 clones, with high time complexity. Abdul-El-Hafiz et al. [48] use a metric-based data mining approach. They use a fractal-based clustering algorithm. This technique uses four steps. 1) Pre-processing the input source file. 2) Extracting all fragments for analysis and related metrics. 3) Partitioning the set of fragments into a small number of clusters of three types using fractal clustering. Primary clusters cover Type-1 and Type-2 clones, Intermediate clusters cover Type-3 clones, and a singleton cluster is not a clone. 4) Post-processing, the extracted clone classes from the primary cluster. This technique uses eight metrics to detect each type of function. MCD Finder by Kanika et al. [29] use a metric-based approach to develop the MCD Finder for Java. Their tool performs a metric calculation on the Java bytecode instead of directly on the source code. This approach consists of three phases. 1) The Java source code is compiled to make it adaptable to requirement of the tool. 2) The computation phase computes metrics that help detect potential clones. This approach uses 9 metrics [29] for each function.

1) Number of calls from a function,
2) Number of statements in a function,
3) Number of parameters passed to a function,
4) Number of conditional statements in a function,
(5) Number of non-local variables inside a function,
(6) Total number of variables inside a function,
(7) Number of public variables inside a function,
(8) Number of private variables inside a function, and
(9) Number of protected variables inside a function.

The calculated metrics are stored in a database and mapped onto Excel sheets. 3) The measurement phase performs a comparison on the similarity of the metric values.

5.2.6 Summary of Syntactical Approaches. In Table 7, a summary of syntactical techniques considering several properties are provided. The first kind of syntactical approach is the tree-based technique. One such system, CloneDr, finds sub-tree clones with limitations as follows. 1) It has difficulty performing near-miss clone detection, but comparing trees for similarity solves it to some extent. 2) Scaling up becomes hard when the software system is large and the number of comparisons becomes very large. Splitting the comparison sub-trees with hash values solves this problem. The parser also parses a full tree. Wahler et al.’s approach detects Type-1 and Type-2 clones only. The clones are detected with a very low recall. Deckard detects significantly more clones and is much more scalable than Wahler et al.’s technique because Deckard uses characteristic vectors and efficient vector clustering techniques. Koschke et al. show that suffix-tree clone detection scales very well since a suffix tree finds clones in large systems and reduces the number of subtree comparisons.

The second kind of the syntactical approach is the metric-based technique. Mayrand et al.’s approach does not detect duplicated code at different granularities. Kontogiannis et al.’s approach works only at block level, it cannot detect clone fragments that are smaller than a block, and it does not effectively deal with renamed variables or work with non-contiguous clones code. Kodhai et al.’s approach only detects Type-1 and Type-2 clones. The limitation of Abdul-El-Hafiz et al.’s technique is that Type-4 clones cannot be detected. The MCD Finder is efficient in detecting semantic clones because byte code is platform independent whereas CCFinder cannot detect semantic clones. However, even MCD Finder tool cannot detect all clones.

Syntactical techniques have limitations as follows. 1) Tree-based techniques do not handle identifiers and literal values for detecting clone in ASTs. 2) Tree-based techniques ignore identifier information. Therefore, they cannot detect reordered statement clones. 3) Metric-based techniques need a parser or PDG generator to obtain metrics values. 4) Two code fragments with the same metric values may not be similar code fragments based on metrics alone.

5.2.7 Semantic Approaches. A semantic approach, which detects two fragments of code that perform the same computation but have differently structured code, uses static program analysis to obtain more accurate information similarity than syntactic similarity. Semantic approaches are categorized into two kinds of techniques. The two kinds are Graph-based techniques and Hybrid techniques. Several semantic techniques from the literature are shown in Table 8. In this section, we discuss Komondoor and Horwitz [22]. Duplix [31] by Krinke, GPLAG [32] by Liu et al., Higo and Kusumoto [49], Hummel et al. [59], Funaro et al. [17], and Agrawal et al. [18].

5.2.7.1 Graph-based Clone Detection Techniques. A graph-based clone detection technique uses a graph to represent the data and control flow of a program. One can build a program Dependency Graph (PDG) as defined in Definition 1 in Section 2. Because a PDG includes both control flow and data flow information as given in Definitions 5 and 6, respectively, one can detect semantic clones using PDG. Clones can be detected as isomorphic subgraphs. In PDG edges represent the data and control dependencies between vertices which repeat lines of code, in PDGs.

Definition 5. (Control Dependency Edge). There is a control dependency edge from a vertex to a second program vertex in a Program Dependency Graph if the truth of the condition controls whether the second vertex will be executed.

Definition 6. (Data Dependency Edge). There is a data dependency edge from program vertex var1 to var2 if there is some variable such that:

- var1 may be assigned a value, either directly or indirectly through pointers.
- var2 may use the value of the variable, either directly or indirectly through pointers.
- There is an execution path in the program from the code corresponding to var1 to the code corresponding to var2 along which there is no assignment to variable var2.

Komondoor and Horwitz Komondoor and Horwitz [22] use program slicing [38] to find isomorphic PDG subgraphs and code clones. As mentioned earlier, nodes in a PDG represent statements and predicates, and edges represent data and control dependences as shown in Example 1 and Figure 3. The slicing clone detection algorithm performs three steps. 1) Find pairs of clones by partitioning all PDG nodes into equivalence classes, where any two nodes in the same class are matching nodes. 2) Remove subsumed clones. A clone pair subsumes another clone pair if and only if each element of the clone pair is a subset of another element from another clone pair. So, subsumed clone pairs need to be removed. 3) Combine pairs of clones into larger groups using transitive closure. Duplix by Krinke finds maximal similar PDG subgraphs with high precision and recall. Krinke’s approach is similar to the Komondoor and Horwitz approach although [22] starts from every pair of matching nodes and uses sub-graphs that are not maximal and are just subtrees unlike the ones in [31]. The PDG used by Krinke is similar to AST and the traditional PDG. Thus, the PDG contains vertices and edges that represent components of expressions. It also contains immediate (control) dependency edges. The value dependency edges represent the data flow between expression components. Another edge, the reference dependency edge, represents the assignments of values to variables.

GPLAG by Liu et al. Liu et al. [32] develop an approach to detect software plagiarism by mining PDGs. Their tool is called GPLAG. Previous plagiarism detection tools were only partially sufficient for academic use in finding plagiarized programs in programming classes. These tools were based on program token methods such as JPlag [33] and are unable to detect disguised plagiarized code well. Plagiarism disguises may include the following.

1) Format alteration such as inserting and removing blanks or comments. 2) Variable renaming where variables names may be changed without affecting program correctness. 3. Statement reordering, when some statements may be reordered without affecting the results. 4) Control replacement such as a for loop can be substituted by a while loop and vice versa. In addition, an for (int i=0; i<10; i++) {a=b-c;} block can be replaced by while (i<10) {a=b-c; i++; }. 5) Code Insertion, where additional
code may be inserted to disguise plagiarism without affecting the results.

Scorpio by Higo and Kusumoto [62] propose a PDG-based incremental two-way slicing approach to detect clones, called Scorpio. Scorpio has two processes: 1) Analysis processing: The inputs are source code files and the output is a database. PDGs are generated from the algorithms of source files. Then, all the edges of the PDGs are extracted and stored in a database. 2) Detection processing: The inputs are source files and the database, and the output is a set of clone pairs. First, a user provides file names and the edges are retrieved for the given files from the database. Finally, the clone pairs are detected by the detection model. This approach detects non-contiguous clones while other existing incremental detection approaches cannot detect non-contiguous clones. The approach also has faster speed compared to other existing PDG based clone detection approaches.

5.2.7.2 Hybrid Clone Detection Techniques. A hybrid clone detection technique uses a combination of two or more techniques. A hybrid approach can overcome problems encountered by individual tools or techniques. ConQAT Hummel et al. [19] use a hybrid and incremental index based technique to detect clones and implement a tool called ConQAT. Code clones are detected in three phases. 1) Preprocessing, which divides the source code into tokens, normalizes the tokens to remove comments or renamed variables. All normalized tokens are collected into statements. 2) Detection, which looks for identical sub-strings. 3) Post-processing, which creates code cloning information looking up all clones for a single file using a clone index. Statements are hashed using MD5 hashing [62]. Two entries with the same hash sequence are a clone pair. The approach extracts all clones for a single file from the index and reports maximal clones. Funaro et al. Funaro et al. [17] propose a hybrid technique that combines a syntactic approach using an abstract syntax tree to identify potential clones with a textual approach to avoid false positives. The algorithm has four phases: 1) Building a forest of ASTs. 2) Serializing the forest and encoding into a string representation with an inverse mapping function. 3) Seeking serialized clones. 4) Reconstructing clones. Agrawal et al. Agrawal et al. [18] present a hybrid technique that combines token-based and textual approaches to find code cloning to extend Boreas [9], which cannot detect Type-3 code clones. The token approach can easily detect Type-1 and Type-2 code clones. The textual approach can detect Type-3 code clones that are hard to detect with the token approach. The technique has three phases. 1) The pre-processing phase removes white-spaces and blank lines. Declarations of variables in a single line are combined into tokens. The technique has three phases. 2) The transformation phase breaks the source code into tokens and detects Type-1 and Type-2 code clones. 3) The match detection phase finds code clones using a matrix representation and then replaces each token with an index value. Then a

<table>
<thead>
<tr>
<th>Author/Tool</th>
<th>Transformation</th>
<th>Code Representation</th>
<th>Comparison Method</th>
<th>Complexity</th>
<th>Granularity</th>
<th>Types of Clone</th>
<th>Language Independence</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CloneDr</td>
<td>Parse to AST</td>
<td>AST</td>
<td>Tree matching</td>
<td>(O(n)) where (n) is number of AST nodes</td>
<td>AST node</td>
<td>Type-1, Type-2</td>
<td>Needs parser</td>
<td>Clone pairs</td>
</tr>
<tr>
<td>Wahlin</td>
<td>Parse to AST</td>
<td>AST</td>
<td>Frequent n-gram</td>
<td>N/A</td>
<td>Line</td>
<td>Type-1, Type-2</td>
<td>Needs parser</td>
<td>AST nodes</td>
</tr>
<tr>
<td>Koschke</td>
<td>Parse to AST</td>
<td>AST</td>
<td>AST</td>
<td>N/A</td>
<td>Tokens</td>
<td>Type-1, Type-2</td>
<td>Needs parser</td>
<td>Text</td>
</tr>
<tr>
<td>Jiang et al.</td>
<td>Parse to parse tree then to a set of vectors.</td>
<td>Vectors</td>
<td>Locality-sensitive hashing Tree-Matching algorithm (LSM)</td>
<td>(O(\sqrt{n}\sqrt{d})) where (n) is number of input nodes</td>
<td>Vectors</td>
<td>Type-1, Type-2</td>
<td>Needs a parser</td>
<td>Text</td>
</tr>
<tr>
<td>Hotta et al.</td>
<td>Parse source code to extract blocks using IDT</td>
<td>Hashed blocks</td>
<td>Group blocks based on hash values</td>
<td>N/A</td>
<td>Blocks</td>
<td>Type-1, Type-2</td>
<td>Needs parser</td>
<td>Clone pairs, Clone classes</td>
</tr>
<tr>
<td>Meryand et al.</td>
<td>Parse to AST then (IRL)</td>
<td>Metrics</td>
<td>21 function metrics</td>
<td>Polynomial complexity</td>
<td>Metrics for each function</td>
<td>Type-1, Type-2, Type-3</td>
<td>Needs Damus tool</td>
<td>Clone pairs</td>
</tr>
<tr>
<td>Kontogiannis et al.</td>
<td>Transform to feature vectors</td>
<td>Feature vectors</td>
<td>Use numerical comparisons of metric values and dynamic programming (DP) using minimum edit distance</td>
<td>(O(n^2)) for Nave approach and (O(nm)) for DP-model</td>
<td>Metrics of a begin-end block</td>
<td>Type-1, Type-2, Type-3</td>
<td>Needs a parser and an additional tool</td>
<td>Clone pairs</td>
</tr>
<tr>
<td>Kodhas et al.</td>
<td>Remove white-spaces, comments; mapping and pre-process statements</td>
<td>Metrics</td>
<td>The string matching/textual comparison</td>
<td>N/A</td>
<td>Functional</td>
<td>Type-1, Type-2</td>
<td>Needs a parser</td>
<td>Clone pairs and clusters</td>
</tr>
<tr>
<td>Abdal El-Halid et al.</td>
<td>Preprocess and extract Metrics</td>
<td>Metrics</td>
<td>Data mining clustering algorithm and fractal clustering</td>
<td>(O(M*\log(M))) where (M) is the size of data set</td>
<td>Functional</td>
<td>Type-1, Type-2, can be Type-3</td>
<td>A language independent</td>
<td>Clone Classes</td>
</tr>
<tr>
<td>Kanika et al.</td>
<td>Calculate metrics of Java programs</td>
<td>Metrics</td>
<td>Use 3-phase comparison algorithm, Adaptation, Computation, and Measurement Phases</td>
<td>N/A</td>
<td>Metrics of Java byte code</td>
<td>Type-1, Type-2, Type-3</td>
<td>Needs compiler</td>
<td>Output mapped into Excel sheets</td>
</tr>
</tbody>
</table>
textual approach looks for the same string values line-by-line. In this phase, Type-3 code clones are detected. 4) The filtering phase removes false positives.

5.2.8 Summary of Semantic Approaches. In Table 8, a summary of semantic techniques is provided. The first kind of semantic approaches includes PDG-based techniques. The approach by Komondoor and Horwitz needs a tool to generate the PDG subgraph. But, the major benefit of Komondoor and Horwitz tool is that it can detect gapped clones. Komondoor and Horwitz and Duplix detect semantically robust code clones using PDG for procedure extraction, which is a program transformation that can be used to make programs easier to understand and maintain. Duplix cannot be applied to large systems and is very slow. Tools that do not use PDG can be effectively confused by statement reordering, replacing, and code insertion. Since PDG is robust to the disguises that confuse other tools, GPLAG is more effective and efficient than these tools. GPLAG has a limitation that the computational complexity increases exponentially with the size of the software code. Scorpio by Higo and Kusumoto detects non-contiguous clones while other incremental detection approaches cannot do so. It also has faster speed than other PDG based clone detection approaches.

The second kind of syntactical approaches is represented by hybrid techniques. Hummel et al. use an approach similar to ConQAT, but it is different because Hummel et al. use graph-based data-flow models. These two approaches can be combined to speed up clone retrieval. Funaro et al. detect Type-3 clones. They also use a textual approach on the source code to remove uninteresting code. Agrawal et al. can detect clones for code written in C only. Semantic techniques have limitations as follows. 1) PDG-based techniques are not scalable to large systems. 2) PDG-based techniques need a PDG generator. 3) Graph matching is expensive in PDG-based techniques.

6. EVALUATION OF CLONE DETECTION TECHNIQUES

In order to choose the right technique for a specific task, several evaluation metrics can be used. A good technique should show both high recall and precision. In Table 9 and 10, we provide comparing evaluations of tools and summary of evaluation approaches respectively. Some evaluation metrics are discussed below.

6.1 Precision and Recall

Precision and recall are the two most common metrics used to measure the quality of a clone finding program. Precision refers to the fraction of candidate clones returned by the detection algorithm that are actual clones, whereas recall refers to the fraction of relevant candidate clones returned by the detection algorithm. High precision means that the candidate clones are mostly actual code clones. Low precision means that many candidate clones are not real code clones. High recall means most clones in the software have been found. Low recall means most clones in the software have not been found. Precision and recall are calculated as shown in Figure 7 and Equations (8) and (9) respectively:

\[
\text{Precision} = \frac{CC}{AC} \times 100
\]

\[
\text{Recall} = \frac{CC}{PC} \times 100
\]

where CC is the number of all correct clones, AC is the number of all found clones, and PC is the number of clones that exist in the code. A perfect clone detection algorithm has recall and precision values that are both 100%.

6.1.1 Precision. A good tool detects fewer false positives, which means high precision. Line-based techniques detect clones of Type-1 with high precision. There are no returned false positives and the precision is 100%. In contrast, token-based approaches return many false positives because of transformation and/or normalization. Tree-based techniques detect code clones with high precision because of structural information. Metric-based techniques find duplicated code with medium precision due to the fact that two code fragments may not be the same but have similar metric values. Finally, PDG-based techniques detect duplicated code with high precision because of both structural and semantic information.

6.1.2 Recall.. A good technique should detect most or all of the duplicated code in a system. Line-based techniques find only exact copy or Type-1 clones. Therefore, they have low recall. Token-based techniques can find most clones of Type-1, Type-2 and Type-3. So, they have high recall. A tree-based technique does not detect any type of clones, but with the help of other techniques clones can be detected. Metric-based techniques have low recall whereas PDG-based techniques cannot detect all of clones.

6.2 Portability

A portable tool is good for multiple languages and dialects. Line-based techniques have high portability but need a lexical analyzer. Token-based techniques need lexical transformation rules. Therefore, they have medium portability. Metric-based techniques need a parser or a PDG generator to generate metric values. They have low portability. Finally, PDG-based techniques have low portability because they need a PDG-generator.

6.3 Scalability

A technique should be able to detect clones in large software systems in a reasonable time using a reasonable amount of memory. Scalability of text-based and tree-based techniques depends on the comparison algorithms. Token-based techniques are highly scalable when they use a suffix-tree algorithm. Metrics-based techniques are also highly scalable because only metric values of begin-end blocks are compared. PDG-based techniques have low scalability because subgraphs matches are expensive.
Table 8. Summary of Semantic Approaches.

<table>
<thead>
<tr>
<th>Author/Tool</th>
<th>Transformation</th>
<th>Code Represen-tation</th>
<th>Comparison Method</th>
<th>Complexity</th>
<th>Granularity</th>
<th>Types of Clone</th>
<th>Language In-depencence</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Komorowski and Horwitz[22]</td>
<td>PDGs using CodeSurfer</td>
<td>PDGs</td>
<td>Isomorphic PDG subgraph matching using backward slicing</td>
<td>N/A</td>
<td>PDG node</td>
<td>Type-3, Type-4</td>
<td>Needs tool for converting source code to PDGs</td>
<td>Clone pairs and Clone Classes</td>
</tr>
<tr>
<td>Duplica[11]</td>
<td>To PDGs</td>
<td>PDGs</td>
<td>K-length path algorithm</td>
<td>Non-polynomial complexity</td>
<td>PDG Sub-graphs</td>
<td>Type-1, Type-4</td>
<td>Needs tool for converting source code to PDGs</td>
<td>Clone Classes</td>
</tr>
<tr>
<td>CPLAG[12]</td>
<td>Using CodeSurfer</td>
<td>PDGs</td>
<td>Isomorphic PDG subgraph matching algorithm</td>
<td>NP-Complete</td>
<td>PDG Node</td>
<td>Type-1, Type-2, Type-3</td>
<td>Needs tool for converting source code to PDGs</td>
<td>Plagiarism pair of programs</td>
</tr>
<tr>
<td>Higo and Kusumoto[23]</td>
<td>To PDGs</td>
<td>PDGs</td>
<td>Code Clone Detection Module</td>
<td>N/A</td>
<td>Edges</td>
<td>Type-3</td>
<td>Needs tool for converting source code to PDGs</td>
<td>Clone Pairs/Files</td>
</tr>
<tr>
<td>ConQAT[29]</td>
<td>Splits source code into tokens and removes comments and variable names. Then normalized tokens are grouped into statements</td>
<td>Tokens</td>
<td>Suffix-tree-based algorithm then using Index-based clone detection algorithm</td>
<td>OR(\log N) where ( f ) is the number of statements and ( N ) is the number of stored tuples</td>
<td>Substrings</td>
<td>Type-1, Type-2</td>
<td>Language independent</td>
<td>Text clones</td>
</tr>
<tr>
<td>Funaro et al.[31]</td>
<td>Parsed to AST, then Serialized AST</td>
<td>AST</td>
<td>Textual comparison</td>
<td>N/A</td>
<td>Specific parts of AST</td>
<td>Type-1, Type-2, Type-3</td>
<td>Needs parser</td>
<td>String clones</td>
</tr>
<tr>
<td>Agrawal et al.[32]</td>
<td>To tokens</td>
<td>Tokens</td>
<td>Line-by-line textual comparison</td>
<td>N/A</td>
<td>Indexed tokens</td>
<td>Type-1, Type-2, Type-3</td>
<td>Needs lexer</td>
<td>Text clones</td>
</tr>
</tbody>
</table>

Table 9. Comparing Evaluations of Tools. It is difficult to compare results of clone detection tools in a fair manner. The results given in this table from various sources are pieced together.

<table>
<thead>
<tr>
<th>Author/Tool</th>
<th>Recall</th>
<th>Precision</th>
<th>F-measure</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duplica[11]</td>
<td>56% - 81.5%</td>
<td>3.1% - 9.3%</td>
<td>5.95% - 16.04%</td>
<td>5</td>
</tr>
<tr>
<td>Duploc[1]</td>
<td>4.5% - 81.5%</td>
<td>3.5% - 12.8%</td>
<td>5.91% - 22.13%</td>
<td>55</td>
</tr>
<tr>
<td>NICAD[4]</td>
<td>13.1% - 76.3%</td>
<td>1.6% - 52.8%</td>
<td>3.13% - 50.07%</td>
<td>11</td>
</tr>
<tr>
<td>SDD[12]</td>
<td>≈ 24%</td>
<td>≈ 22%</td>
<td>≈ 22.06%</td>
<td>13</td>
</tr>
<tr>
<td>CC-Finder[4]</td>
<td>44.5% - 100%</td>
<td>0.8% - 6.6%</td>
<td>1.58% - 12.33%</td>
<td>4</td>
</tr>
<tr>
<td>CP-Miner[8]</td>
<td>≈ 48%</td>
<td>≈ 41%</td>
<td>≈ 44.22%</td>
<td>4</td>
</tr>
<tr>
<td>Borack[4]</td>
<td>≈ 95%</td>
<td>≈ 5%</td>
<td>≈ 9.5%</td>
<td>9</td>
</tr>
<tr>
<td>FRISC[1]</td>
<td>≈ 79% - 98%</td>
<td>≈ 10% - 50%</td>
<td>≈ 17.7% - 66.2%</td>
<td>65</td>
</tr>
<tr>
<td>CDSW[1]</td>
<td>9.27% - 70.63%</td>
<td>4.43% - 49.97%</td>
<td>8.09% - 28.70%</td>
<td>66</td>
</tr>
<tr>
<td>Baxter, et al.[1]</td>
<td>14.9% - 48.1%</td>
<td>6% - 40.3%</td>
<td>8.68% - 32.5%</td>
<td>13</td>
</tr>
<tr>
<td>Wahrer et al.[5]</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>14</td>
</tr>
<tr>
<td>Koschke et al.[5]</td>
<td>≈ 33%</td>
<td>N/A</td>
<td>N/A</td>
<td>14</td>
</tr>
<tr>
<td>Jiang et al.[3]</td>
<td>3.4% - 85.4%</td>
<td>2.2% - 6.6%</td>
<td>4.29% - 8.91%</td>
<td>28</td>
</tr>
<tr>
<td>Hotta et al.[8]</td>
<td>23.3% - 50.1%</td>
<td>9.9% - 22.8%</td>
<td>14% - 23.21%</td>
<td>67</td>
</tr>
<tr>
<td>Mayrand et al.</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>20</td>
</tr>
<tr>
<td>Kontogiannakis et al.</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>20</td>
</tr>
<tr>
<td>Kodhai, et al.[7]</td>
<td>≈ 99%</td>
<td>≈ 100%</td>
<td>≈ 99.50%</td>
<td>19</td>
</tr>
<tr>
<td>Abdul-El-Hafiz et al.[2]</td>
<td>N/A (Difficult to assess)</td>
<td>≈ 52% - 100%</td>
<td>N/A</td>
<td>48</td>
</tr>
<tr>
<td>Kanika et al.</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>49</td>
</tr>
<tr>
<td>Duploc[1]</td>
<td>17.3% - 45.8%</td>
<td>2.9% - 10.5%</td>
<td>5.35% - 17.08%</td>
<td>131</td>
</tr>
<tr>
<td>GPLAG</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>132</td>
</tr>
<tr>
<td>Higo and Kusumoto[23]</td>
<td>≈ 97% - 100%</td>
<td>≈ 96% - 97%(Type-3)</td>
<td>≈ 96.50% - 98.48%</td>
<td>49</td>
</tr>
<tr>
<td>ConQAT</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>43</td>
</tr>
<tr>
<td>Funaro et al.</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>17</td>
</tr>
<tr>
<td>Agrawal et al.</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>18</td>
</tr>
</tbody>
</table>

(1): The displayed detectors results are obtained from Murkami et al.[66], who use netbeans, ant, jcore, swing, wealtab,cook,snns, and postgresql datasets. (2): The results given are obtained from Shaheian and Zou[75], who use EIRC, Spule, and Apache Ant datasets. (3): The results given here are obtained from Dang and Wani[76], who use EIRC dataset. (4): The given results are quoted from Yuan and Guo[9], who use Java SE Development Kit 7 and Linux kernel 2.6.38.6 datasets. (5): The results given are obtained from Falke and Koschke[14], who use bison 1.32, wget 1.5.3, SNNS 4.2, and postgresQL 7.2 datasets. (6): The results are obtained from Hotta et al.[67], who use netbeans, ant, jcore and swing datasets. (7): The given results are obtained from Kodhai et al.[18], who use Wealtab dataset. (8): The results are obtained from Abdul-El-Hafiz et al.[22], who use Wealtab and SNNS datasets. (9): The results are obtained from Higo and Kusumoto[49], who use Ant dataset.
6.4 Clone Relation

Clones reported as clone classes are more useful than clone pairs. Finding clone classes reduces the number of comparisons such as in the case of the NICAD tool [13]. They are more useful for maintenance than clone pairs.

6.5 Comparison Units

There are various levels of comparison units such as source lines, tokens, subtrees and subgraphs. Text-based techniques compare the source code line-by-line, but their results may not be meaningful syntactically. Token-based techniques use tokens of the source code. However, token-based techniques can be less efficient in time and space than text-based techniques because a source line may contain several tokens. Tree-based techniques use tree nodes for comparison units and search for similar trees with expensive comparison, resulting in low recall. Metric-based techniques use metric values for each code fragment but it could be that the metric values for cloned code are not the same. PDG-based techniques use PDG nodes and search for isomorphic subgraphs but graph matching is costly.

6.6 Complexity

Computational complexity is a major concern in code clone detection techniques. A good clone detection technique should be able to detect clones in large systems. The complexity of each technique depends on the comparison algorithm, comparison unit and the type of transformation used.

6.7 Transformation or Normalization

Transformation and normalization remove uninteresting clones and remove comments. Thus, these steps help filter noise and detect near-miss clones. A good tool should filter noise and detect near-miss clones.

7. RELATED AREAS

Code clone detection techniques can help in areas such as clone refactoring or removal, clone avoidance, plagiarism detection, bug detection, code compacting, copyright infringement detection and clone detection in models.

7.1 CLONE REFACTORIZING OR REMOVAL

Code clones can be refactored after they are detected. Code clone refactoring is a technique that restructures existing cloned code without changing its external behavior or functionality. Code clone refactoring improves design, flexibility and simplicity without changing a program’s external behavior. Refactoring or removal is used to improve maintainability and comprehensibility of software [40]. However, Kim et al. [10] show that clone refactoring is not always a solution to improve software quality because of two issues. First, clones are frequently short-lived. Refactoring is not good if there are branches from a block within short distances. Second, long-lived clones that have been modified with other elements in the same class are hard to be removed or refactored. In addition, a bug that can be fixed easily because the source code is easy to understand improves malleability leading to code extensibility.

Goto et al. [70] present an approach based on coupling or cohesion metrics to rank extract methods for merging. They use AST differencing to detect syntactic differences between two similar methods using slice-based cohesion metrics. This method helps developers or programmers in refactoring similar methods into cohesive methods.

Meng et al. [71] design and implement a new tool called RASE. RASE is an automated refactoring tool that consists of four different types of clone refactoring approaches: use of extract methods, use of parameterized types, use of form templates, and adding parameters. RASE performs automated refactoring by taking two or more methods to perform systematic edits to produce the target code with clones removed.

Fontana et al. [73] conclude that removing duplicated code leads to an improvement of the system in most cases. They analyse the impact of code refactoring on eight metrics, which five metrics on package level and three metrics on class level.

At Package level [73]

(1) LOC: Line of code at Package level,
(2) NOM: Number of methods at Package level,
(3) CC: Cyclomatic complexity at Package level,
(4) CF: Coupling factor at Package level, and,
(5) DMS: Distance from the main sequence at Package level.

At Class level [73]

(1) WMC: Weighted method complexity,
(2) LCOM: Lack cohesion in methods, and,
(3) RFC: Response for a Class.

They calculate the metrics value and observe that the quality metrics are improved in class level. Therefore, refactoring code improves the quality of specific classes.

7.2 Types of code refactoring

There are a number of different types of code refactoring approaches that can be performed.

(1) Extract Class. When two classes are identical or have similar subclasses, they can be refactored by creating a new class, moving the relevant fields and methods from the old class into the new class.
Table 10. Summary of Evaluation Approaches.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Precision</th>
<th>Recall</th>
<th>Scalability</th>
<th>Portability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text-based</td>
<td>High</td>
<td>Low</td>
<td>Depends on comparison algorithm</td>
<td>High</td>
</tr>
<tr>
<td>Token-based</td>
<td>Low</td>
<td>High</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>Tree-based</td>
<td>High</td>
<td>Low</td>
<td>Depends on comparison algorithm</td>
<td>Low</td>
</tr>
<tr>
<td>Metric-based</td>
<td>Medium</td>
<td>Medium</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>PDG-based</td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
<td>Low</td>
</tr>
</tbody>
</table>

Fig. 8. Simple Example of Pull Up Method.

Fig. 9. Simple Example of Add parameter in Method.

(2) Extract Method. The easiest way of refactoring a clone is to use the extract method of refactoring. The extract method extracts a fragment of source code and redefines it as a new method. In this type, code clones can be removed as shown in Figure 8.

(3) Rename Methods. Renaming a method does not reveal the method’s purpose or change the name of the method.

(4) Push Down Method. Moving a method from a superclass into a sub-class.

(5) Pull Up Method. Moving a method from a subclass into a super-class. This type can remove the code clones as shown in Figure 8.

(6) Add Parameter. When a new method or function needs more information from its caller, the code can be refactored as shown in Figure 8.

(7) Replace Constructor With Factory. This can be used when there is an object that is created with a type code and later it needs sub-classes. Constructors can only return an instance of the object. So, it is necessary to replace the constructor with a factory method as shown in Figure 11.

Fig. 10. Example of Replace Constructor With Factory.

Juillerat et al. [42] propose a new algorithm for removing code clone using Extract refactoring for Java. Kodhai et al. [40] use Rename, Add Parameter, and Replace Constructor With Factory methods after detecting functions or methods in C or Java applications to remove or refactor code clones. Choi et al. [41] propose a method to combine clone metric values to extract clone sets from CCFinder output for refactoring. They use the metric graph of Gemini [63] for analyzing the output of CCFinder.

7.3 CLONE AVOIDANCE

Two approaches are discussed to deal with cloning, how to detect clones and how to remove clones. The third approach is avoidance, which tries to disallow the creation of code clones in the software right from the beginning. Legue et al. [43] use a code clone detection tool in two ways in software development. The first way uses code clone detection as preventive control where any added code fragment is checked to see if it is a duplicated version of any existing code fragment before adding it to the system. The second way, problem mining, searches for the modified code fragment in the system for all similar code fragments.

7.4 Plagiarism Detection

Code clone detection approaches can be used in plagiarism detection of software code. Dup [68] is a technique that is used for finding near matches of long sections of software code. JPlag [33] is another tool that finds similarities among programs written in C, C++, Java, and Scheme. JPlag compares bytes of text and program structure. Yuan et al. [44] propose a count-based clone detection technique called CMCD. CMCD has been used to detect plagiarisms in homeworks of students.

7.5 Bug Detection

Code clone detection techniques can also help in bug detection. CP-Miner [3] has been used to detect bugs. Higo et al. [35] propose an approach to efficiently detect bugs that are caused by copy-paste programming. Their algorithm uses existing detection tools such as CCFinderX [69].
7.6 Code Compacting

Code size can be reduced by using code clone detection techniques and replacing common code using code refactoring techniques [5].

7.7 Copyright Infringement

Clone detection tools can easily be adapted to detect copyright infringement [8].

7.8 Clone Detection in Models

Model-based development can also use clone detection techniques to detect duplicated parts of models [46], Deissenboeck et al. [47] propose an approach for an automatic clone detection using large models in graph theory.

8. OPEN PROBLEMS IN CODE CLONE DETECTION

There is no clone detection technique which is perfect in terms of precision, recall, scalability, portability and robustness. Therefore, it is necessary to come up with new clone detection approaches that can do better by overcoming some of the limitations of existing techniques. There is hardly any tool that can detect Type-3 and Type-4 clones. Type-4 clones require to solve an undecidable problem [64]. Clone detection technique can be improved by combining several different types of methods or reimplementing systems using a different programming language. This presents new challenges for software maintenance, refactoring and clone management.

It is hard to determine which is the best tool for detection because every tool has its strengths and weaknesses. Since text-based and token-based techniques have high recall and AST-based techniques have high precision, these techniques may be merged in a tool to get high recall and precision results. A PDG-based technique detects only Type-3 clones; this technique may be extended to detect Type-1 and Type-2 clones besides Type-3 clones.

Type-1 and Type-2 clones are easier to detected than Type-3 clones. Sequence alignment algorithms with gaps may potentially be used to detect Type-3 clones. To detect plagiarism in program code, text similarity measures and local alignment for high scalability may be used.

9. CONCLUSION

Software clones occur due to reasons such as code reuse by copying pre-existing fragments, and repeated computations using duplicated functions with slight changes in the used variables, data structures or control structures. After introducing code clones, common types of clones, phases of clone detection, we have discussed at length code clone detection techniques, covering a number of techniques in detail. Recent code clone detectors and tools that have not been discussed in previous survey papers are covered. This survey stands apart from other prior surveys in that it organizes the discussion in terms of a classification scheme, categorizes clone detection techniques, under classes and subclasses, provides a level of detail not found in other surveys to facilitate a newcomer to the field to get started and get into the field full-speed, provides extensive comparison of the methods so that one can choose right method needed for one’s needs as well as develop new methods that may overcome drawbacks of existing methods.

10. REFERENCES


[52] Han, Jiawei. Data Mining: Concepts and Techniques. (2006).